Список вопросов к экзамену по дисциплине
Программирование на языке Джава
зима 2024-2025 год

1. Введение в Java. Основные характеристики языка, сферы применения, история создания. Экосистема языка JAVA. JDK, JRE, JVM.
2. Основные платформы Java. Java SE, Java EE, Java ME, их особенности и области применения.
3. Виртуальные машины и их роль в JAVA. Архитектура JVM. Основные компоненты: Class Loader, Execution Engine, Garbage Collector.
4. Компиляция Java-программ. Различия между JIT (Just-in-Time) и AOT (Ahead-of-Time) компиляцией. Преимущества и недостатки.
5. Модель памяти в Java. Основные области памяти JVM: куча (Heap) и стек (Stack), их назначение и различия. Как распределяются объекты и примитивные данные в этих областях? Что такое Young Generation, Old Generation и Metaspace? Как работа сборщика мусора влияет на управление памятью?
6. Основные парадигмы программирования в Java. Объектно-ориентированное, функциональное, многопоточное программирование.
7. Виртуальные машины и их роль в JAVA. Особенности стандартной HotSpot JVM. GraalVM и другие сторонние виртуальные машины для Java. Основные преимущества и возможности сторонних виртуальных машин.
8. Компиляция и запуск проекта на Java. Обеспечение переносимости кода на различные платформы. Понятие промежуточного байт-кода и его роль в переносимости программ. Чем отличаются методы компиляции JIT (Just-In-Time) и AOT (Ahead-of-Time), и как они влияют на производительность и переносимость?
9. Современный инструментарий разработчика Java. Популярные IDE и их возможностей для написания, отладки и сборки кода. Основные системы сборки и их роль в управлении проектами на JAVA. Контроль версий с использованием Git и интеграция с платформами хостинга ИТ-проектов. Использование Docker и Kubernetes для контейнеризации и оркестрации приложений. Инструменты CI/CD для автоматизации сборки, тестирования и деплоя JAVA приложений.
10. Современные фреймворки для разработки на Java. Особенности Spring Framework. Основные возможности Hibernate. Основные причины использования данных фреймворков при разработке на JAVA.
11. Объектная модель Java. Основные принципы объектной модели в Java: классы, объекты, интерфейсы, наследование и инкапсуляция. Класс Object и методы, которые он предоставляет.
12. Пакеты в Java. Основное предназначение. Структура, организация, использование в программировании (импорт пакетов).
13. Синтаксис и лексика Java. Основные элементы лексики языка: ключевые слова, идентификаторы, литералы, комментарии, операторы и разделители. Правила именования идентификаторов. Соглашения по оформлению кода.
14. Типы данных в Java. Примитивные типы данных, объявление и присваивание переменных. Отличия примитивных типов данных от ссылочных.
15. Типы данных в Java. Ссылочные типы данных, объявление и присваивание переменных. Отличия ссылочных типов данных от примитивных. Роль классов-оберток (Wrapper Classes) для работы с примитивами.
16. Константы в Java. Понятие констант и их объявление с использованием ключевого слова final. Основные правила и соглашения по именованию констант. Примеры создания констант для примитивных типов данных и строк. Как константы помогают обеспечить неизменность данных и улучшают читаемость кода?
17. Ключевое слово var в Java. Особенности использования var для объявления локальных переменных. Как происходит неявное выведение типа переменной компилятором? Ограничения на использование var: недопустимость для полей класса, параметров методов и возвращаемых типов.
18. Соглашения по оформлению кода Java. Java Code Conventions и её значение для совместной работы.

19. Класс и экземпляры класса. Что такое класс в Java и как происходит создание объектов (инстанцирование) с использованием ключевого слова new? Примеры создания и использования экземпляров класса.
20. Записи (Records) в Java. Какие возможности они предоставляют и в чем их отличие от обычных классов? Примеры использования записей.
21. Запечатанные (Sealed) классы. Как они ограничивают наследование и для чего используются?
22. Инкапсуляция в Java. Понятие инкапсуляции как механизма защиты данных и управления доступом к ним. Реализация инкапсуляции с использованием модификаторов доступа (private, protected, public, package-private). Роль геттеров и сеттеров в обеспечении контроля за изменением данных объекта. Примеры нарушения инкапсуляции и способы предотвращения этих ошибок.
23. Модификаторы доступа. Какие уровни доступа существуют в Java? Как модификаторы доступа используются для контроля видимости классов, полей и методов?
24. Модификатор final. Применение final к переменным, методам и классам. Как он предотвращает изменения данных, поведение методов и наследование?
25. Конструкторы в Java. Понятие конструктора и его роль в создании объектов. Различия между конструктором и методом. Типы конструкторов. Как реализовать перегрузку конструкторов?
26. Конструкторы в Java. Понятие конструктора и его роль в создании объектов. Использование ключевого слова this для вызова одного конструктора из другого. Особенности работы конструкторов в наследовании, вызов конструктора родительского класса через super.
27. Блоки инициализации. Виды блоков инициализации: статические и нестатические. Их роль в подготовке объектов и классов. Примеры использования блоков для сокращения повторяющегося кода.
28. Статические блоки инициализации. Примеры и использование статических блоков для выполнения кода при загрузке класса. Их роль в инициализации общих данных.
29. Модификатор static. Особенности использования static для полей, методов и блоков. Различия между статическими и нестатическими членами класса. Примеры применения для создания общих ресурсов.
30. Ключевое слово this. Использование this для доступа к полям и методам объекта, вызова других конструкторов и передачи текущего объекта. Примеры решения конфликтов имен с помощью this.
31. Концепция неизменяемых классов. Что делает класс неизменяемым? Использование final для предотвращения изменений. Примеры создания неизменяемых объектов.
32. Создание объектов. Отличие фабричных методов от стандартного создания объектов с использованием new. Примеры использования фабричных методов.
33. Рефлексия в Java. Возможности рефлексии для создания объектов и вызова методов во время выполнения. Примеры использования рефлексии для создания объектов.
34. Жизненный цикл объектов в JAVA. Роль сборщика мусора в управлении памятью. Примеры оптимизации работы объектов в Java.
35. Инициализация переменных в JAVA. Способы инициализации переменных: по умолчанию, в конструкторах, через блоки инициализации. Примеры применения.
36. Математические функции. Класс Math в Java и его методы для выполнения вычислений. Примеры использования тригонометрических и экспоненциальных функций в задачах. Нужно ли создавать объект класса Math для использования математических методов.

37. Абстракция и инкапсуляция класса. Понятие абстракции как отделения реализации класса от его использования. Как эти принципы улучшают структурирование кода и его модульность?
38. Отношения между классами. Основные виды отношений между классами: ассоциация, агрегация, композиция, наследование.
39. Ассоциация. Понятие ассоциации как бинарного отношения между классами. Примеры реализации ассоциации в Java. Как ассоциация помогает моделировать взаимодействие объектов?
40. Агрегация и композиция. Понятия агрегации и композиции, их различия. Как они отражают отношения «has-a» между объектами? Примеры реализации агрегации и композиции в проектировании классов.
41. Обработка примитивных типов как объектных. Использование классов-оберток для работы с примитивными типами как с объектами. Примеры преобразования примитивных типов в объекты и обратно.

42. Классы-обертки. Основные возможности классов-оберток: Integer, Double, Boolean и других. Методы для преобразования значений и сравнения объектов. Примеры использования методов parseInt, valueOf и compareTo.
43. Автоматическое преобразование. Что такое автоупаковка (autoboxing) и автораспаковка (unboxing) в Java? Как они автоматически преобразуют значения примитивных типов в объекты и обратно? Примеры использования.
44. Класс String. Понятие неизменяемости(иммутабельности) строк в Java. Как создаются объекты типа String? Примеры работы с методами создания, сравнения и модификации строк.
45. Строки в JAVA. Замена и разделение строк. Методы класса String для замены символов и разделения строк. Примеры работы с методами replace и split.
46. Строки в JAVA. Преобразования между строками и массивами. Как преобразовать строку в массив символов и наоборот? Примеры использования методов toCharArray и valueOf.
47. Строки в JAVA. Класс StringBuilder и StringBuffer. Понятие изменяемых строк. Основные отличия между StringBuilder и StringBuffer. Примеры их использования. Влияние классов StringBuilder и StringBuffer на типобезопасность.
48. Строки в JAVA. Преобразование символов и чисел в строки. Какие методы используются для преобразования чисел, символов и объектов в строки? Примеры работы с методами String.valueOf() и toString().
49. Строки в JAVA. Интернированные строки. Что такое интернированные строки? Как JVM оптимизирует работу с повторяющимися строками? Примеры их использования.
50. Наследование в JAVA. Основные принципы наследования в Java. Что такое суперклассы(родительские) и подклассы(дочерние)? Как наследование помогает переиспользовать код? Примеры реализации наследования.
51. Перегрузка метода в Java (overload). Переопределение метода в Java (override). В чем разница между перегрузкой и переопределением методов.
52. Наследование и отношение is-a. Как наследование реализует отношение «is-a»? Когда использование наследования может быть нецелесообразным? Примеры решений.
53. Ключевое слово super. Роль ключевого слова super в Java. Использование для вызова методов и конструкторов суперкласса. Примеры реализации.
54. Цепочка конструкторов. Понятие цепочки конструкторов. Как вызвать один конструктор из другого с использованием this() и super()? Примеры реализации.
55. Класс Object и его основные методы. Роль класса Object как суперкласса для всех классов в Java. Как метод toString() используется для представления объекта в виде строки? Примеры переопределения метода.
56. Полиморфизм. Понятие полиморфизма в Java. Как переменная супертипа может ссылаться на объект подтипа? Примеры применения полиморфизма для создания гибкого кода.

57. Интерфейсы в Java. Понятие интерфейсов как конструкций для определения общих операций. Основные элементы интерфейсов: константы и абстрактные методы. Примеры использования интерфейсов для создания обобщенных решений.
58. Интерфейсы в Java. Понятие интерфейсов как конструкций для определения общих операций. Особенности интерфейсов, добавленные в JAVA 8 версии. Дефолтные методы в интерфейсах.
59. Интерфейсы в Java. Особенности интерфейсов. Чем интерфейсы отличаются от классов? Как используются ключевые слова interface и implements? Примеры объявления и реализации интерфейсов.
60. Интерфейсы в Java 8 и 9. Новые возможности интерфейсов, такие как default и static методы (Java 8), а также private и private static методы (Java 9). Примеры реализации и применения.
61. Интерфейс Comparable. Как интерфейс Comparable используется для сравнения объектов? Реализация метода compareTo() и его роль в сортировке. Примеры работы с интерфейсом.
62. Интерфейс Comparable для классов стандартной библиотеки JAVA. Как реализован интерфейс Comparable в классах String, Integer и Date? Примеры сравнения объектов с помощью метода compareTo().
63. Интерфейс Comparable для пользовательских классов. Как реализовать интерфейс Comparable для пользовательских классов? Примеры сравнения объектов на основе пользовательских критериев.
64. Интерфейс Cloneable. Понятие клонирования объектов. Как интерфейс Cloneable позволяет клонировать объекты? Ограничения и примеры использования.

65. Метод clone(). Как метод clone(), определенный в классе Object, используется совместно с интерфейсом Cloneable? Примеры работы с клонируемыми объектами.
66. Интерфейсы и абстрактные классы. Основные различия между интерфейсами и абстрактными классами.
67. Понятие абстрактных классов в Java. Что такое абстрактный класс, и как он используется для создания общего базового поведения? Чем отличается абстрактный класс от интерфейса? Примеры объявления и реализации абстрактного класса с абстрактными и конкретными методами.
68. Понятие абстрактных классов в Java. Объявление абстрактных методов. Что такое абстрактный метод, и какие правила нужно соблюдать при его объявлении? Как абстрактные методы помогают подклассам реализовать специфическое поведение? Примеры реализации абстрактных методов в наследуемых классах.
69. Понятие абстрактных классов в Java. Особенности работы с абстрактными классами. Почему абстрактные классы нельзя инстанцировать? Как использовать абстрактный класс как основу для других классов? Примеры создания иерархии классов с базовым абстрактным классом.
70. Ограничение множественного наследования в JAVA. Множественное наследование интерфейсов. Как классы наследуют методы от нескольких интерфейсов.
71. Интерфейсы в Java. Особенности интерфейсов. Интерфейсы и полиморфизм. Как интерфейсы способствуют реализации полиморфизма?

72. Обработка исключительных ситуаций в JAVA. Основные способы и подходы к обработке исключительных ситуаций в JAVA. Иерархия классов исключений в Java. Понятие и структура иерархии исключений. Чем отличаются классы Error, Exception и RuntimeException?
73. Создание и генерация исключений. Как создавать и генерировать исключения с помощью ключевого слова throw? Различия между throw и throws. Примеры создания пользовательских исключений.
74. Обработка исключений. Структура блока try-catch. Как обрабатывать исключения с использованием блоков try-catch? Примеры обработки нескольких исключений и упорядочения блоков catch. Роль объекта исключения (Exception e) в блоке catch.
75. Обработка исключений. Структура блока try-catch. Блок finally и его использование. Основные причины использования. Примеры использования.
76. Обработка исключений. Пропагирование исключений. Как исключения передаются вверх по стеку вызовов? Примеры использования ключевого слова throws в сигнатуре методов.
77. Обработка исключений. Проверяемые и непроверяемые исключения. Какие исключения считаются проверяемыми (checked), а какие - непроверяемыми (unchecked)? Примеры работы с ними. Исключения в популярных фреймворках. Почему большинство исключений в современных фреймворках являются непроверяемыми?
78. Обработка исключений. Использование try-with-resources. Как она упрощает управление ресурсами? Примеры работы.
79. Обработка исключитеьных ситуаций в JAVA. Роль JVM в обработке исключений. Как JVM управляет исключениями, если они не были обработаны? Примеры поведения при неперехваченных исключениях.
80. Перечисления (enums) в Java. Что такое перечисления и как они используются для создания фиксированных наборов значений? Характеристики перечислений. Перечисления и типобезопасность. Примеры их применения.
81. GUI в Java. Что такое GUI (графический пользовательский интерфейс)? Основные пакеты для работы с GUI в Java: AWT и Swing.
82. GUI в Java. Структура GUI в JAVA при реализации через Swing и AWT. Компоненты GUI. Какие элементы составляют графический интерфейс? Примеры кнопок, текстовых полей и других компонентов.
83. AWT (Abstract Window Toolkit). Что такое AWT и как он используется для создания GUI? Примеры простых интерфейсов с использованием AWT.
84. Swing в Java. Как Swing расширяет возможности AWT? Примеры создания интерфейсов с использованием Swing. Паттерн MVC в Swing. Как Swing реализует модель MVC (Model-View-Controller)? Примеры разделения логики, представления и управления в интерфейсе.
85. Структура GUI в Java. Основные компоненты GUI в Swing: контейнеры (JFrame, JPanel, JDialog), компоненты (JButton, JLabel, JTextField) и менеджеры компоновки.

86. Класс JFrame. Что такое окно JFrame, и как использовать его для создания графического интерфейса? Примеры добавления элементов через метод getContentPane().
87. Класс JPanel. Как панель JPanel используется для группировки и управления компонентами? Примеры изменения менеджера компоновки с помощью метода setLayout().
88. Менеджеры компоновки в Java. Роль менеджеров компоновки в управлении размещением компонентов. Примеры использования менеджеров FlowLayout, BorderLayout, GridLayout.
89. Менеджер FlowLayout. Как работает FlowLayout? Примеры настройки выравнивания и промежутков между компонентами.
90. Менеджеры компоновки в Java. Роль менеджеров компоновки в управлении размещением компонентов. Примеры использования менеджеров FlowLayout, BorderLayout, GridLayout.
91. Менеджер FlowLayout. Как работает FlowLayout? Примеры настройки выравнивания и промежутков между компонентами.
92. Менеджер BorderLayout. Как BorderLayout делит контейнер на регионы (NORTH, SOUTH, EAST, WEST, CENTER)? Примеры создания интерфейсов с четкой организацией областей.
93. Менеджер GridLayout. Как компоненты размещаются в сетке с использованием GridLayout? Примеры создания таблиц или форм.
94. Менеджер BoxLayout. Как компоненты размещаются по горизонтали или вертикали с помощью BoxLayout? Примеры последовательного расположения элементов.
95. Границы в Swing. Как использовать границы для улучшения внешнего вида интерфейса? Примеры применения границ.
96. GUI и сбытийная модель в Java. Что такое событийная модель, и как она используется для взаимодействия компонентов через события? Основные элементы событийной модели.
97. Обработка событий в Java. Как источник события, слушатель и обработчик взаимодействуют в событийной модели? Примеры добавления слушателей событий. Модель делегирования событий. Как работает модель делегирования событий?
98. Обработка событий при реализации GUI в JAVA. Классы событий пакета java.awt.event. Какие классы событий предоставляет пакет java.awt.event? Примеры обработки событий мыши и клавиатуры.
99. Обработка событий мыши в JAVA. Как использовать интерфейсы MouseListener и MouseMotionListener для обработки событий мыши? Примеры обработки нажатий и перемещений.
100. Обработка событий клавиатуры в JAVA. Как обрабатывать события клавиатуры с использованием KeyListener? Примеры регистрации слушателей клавиатурных событий.
101. Обобщённое программирование в Java. Понятие обобщённого программирования и его роль в упрощении создания алгоритмов для работы с различными типами данных. История разваития в JAVA. Примеры проектирования универсальных структур данных и алгоритмов.
102. Generics в Java. Реализация обобщенного программирования через Generics. Основные синтаксические конструкции: параметры типов, обобщенные классы и методы. Примеры работы с параметризованными классами и методами. Примущества и недостатки Generics.
103. Коллекции и Generics в Java. Как использование Generics повысило типобезопасность коллекций, таких как ArrayList, HashMap и HashSet? Примеры создания и обработки коллекций с обобщениями.
104. Параметризованные методы. Понятие параметризованных методов в Java. Как они позволяют работать с любыми типами данных? Примеры реализации методов с обобщенными параметрами и их вызова.
105. Generics в Java. Типовые ограничения в Generics. Как задать ограничения на параметры типов с помощью ключевых слов extends и super? Примеры их использования для обеспечения гибкости и безопасности обобщений.
106. Обобщенные интерфейсы. Использование Generics для создания универсальных интерфейсов. Примеры реализации обобщенных интерфейсов и их применения в реальных задачах.
107. Generics в Java. Подстановочные знаки (Wildcards). Как использовать ?, <? extends T> и <? super T> для работы с коллекциями? Примеры их применения.
108. Generics в Java. Стирание типов (Type Erasure). Как информация о Generics удаляется во время компиляции? Примеры преобразования Generics в сырой тип.
109. Коллекции в Java. Понятие коллекций как структур данных для хранения объектов. Основные интерфейсы и классы в Java Collections Framework (JCF). Примеры использования коллекций для хранения и обработки данных.

110. Иерархия коллекций. Структура иерархии коллекций в Java. Основные интерфейсы (Collection, List, Set, Map) и их ключевые особенности. Примеры реализации различных типов коллекций.
111. LinkedList в Java. Особенности класса LinkedList как реализации интерфейса List. Преимущества использования.
112. Коллекции в Java. Понятие коллекций как структур данных для хранения объектов. Основные цели использования коллекций. Роль Iterable в Java Collections Framework.
113. Коллекции в Java. Реализации List - ArrayList. Особенности функционирования ArrayList. Пример использования ArrayList.
114. Коллекции в Java. Создание Generic Collection в Java. Преимущства данного подхода. Примеры.
115. Коллекции и Generics. Использование Generics для типобезопасности в коллекциях. Примеры создания типизированных списков и множеств.
116. ArrayList в Java. Понятие ArrayList как реализации интерфейса List. Основные методы (add, get, remove) для работы со списками. Примеры добавления, удаления и доступа к элементам.

1. Введение в Java. Основные характеристики языка, сферы применения, история создания. Экосистема языка JAVA. JDK, JRE, JVM.

- **Основные характеристики Java:**
- **Объектно-ориентированный:** Java основан на парадигме объектно-ориентированного программирования (ООП), что позволяет создавать модульный, повторно используемый и гибкий код.
- **Платформонезависимый:** Код Java компилируется в байт-код, который может выполняться на любой платформе с установленной виртуальной машиной Java (JVM).
- **Многопоточный:** Java поддерживает многопоточность, что позволяет одновременно выполнять несколько задач в одном приложении.
- **Безопасный:** Java обеспечивает высокий уровень безопасности благодаря своей системе контроля доступа и механизмам защиты от вредоносного кода.
- **Надежный:** Java имеет автоматическое управление памятью (сборщик мусора), что снижает вероятность ошибок, связанных с утечкой памяти.
- **Производительный:** Благодаря JIT-компиляции и оптимизации JVM, Java приложения могут быть очень производительными.
- **Простой:** Синтаксис Java относительно прост для изучения, особенно по сравнению с C или C++.
- **Сферы применения:**
- **Корпоративные приложения:** Разработка крупных, сложных систем для бизнеса (CRM, ERP, банковские системы).
- **Веб-приложения:** Создание динамических веб-сайтов и веб-сервисов (Spring Boot, Jakarta EE).
- **Мобильные приложения (Android):** Разработка приложений для Android-устройств.
- **Игры:** Разработка игр с использованием фреймворков, таких как LibGDX.
- **Научные и инженерные приложения:** Обработка больших данных, моделирование, анализ.
- **Встроенные системы:** Программирование для микроконтроллеров и различных устройств.
- **История создания:**
- Разработан компанией Sun Microsystems (позже Oracle).
- Первая версия выпущена в 1995 году.
- Изначально назывался Oak и предназначался для программирования бытовой электроники.
- Позже переориентирован на разработку веб-приложений.
- Слоган “Write Once, Run Anywhere” (WORA).

•**Экосистема Java:**

•**JDK (Java Development Kit):** Набор инструментов для разработки на Java, включая компилятор, отладчик и другие утилиты.

•**JRE (Java Runtime Environment):** Среда выполнения Java, необходимая для запуска Java-приложений, включает JVM.

•**JVM (Java Virtual Machine):** Виртуальная машина, интерпретирующая байт-код и выполняющая его на целевой платформе.

2. Основные платформы Java. Java SE, Java EE, Java ME, их особенности и области применения.

•**Java SE (Standard Edition):**

•**Особенности:** Базовая платформа для разработки десктопных приложений, консольных программ и общего назначения. Включает основные библиотеки и API.

•**Области применения:** Настольные приложения, консольные утилиты, разработка общего назначения, фундаментальные знания Java.

•**Java EE (Enterprise Edition) / Jakarta EE:**

•**Особенности:** Платформа для разработки крупных корпоративных веб-приложений и веб-сервисов. Включает множество спецификаций для веб-разработки, баз данных, безопасности и т.д.

•**Области применения:** Разработка веб-приложений, веб-сервисов, бизнес-приложений, корпоративных систем, банковское дело, электронная коммерция.

•**Java ME (Micro Edition):**

•**Особенности:** Платформа для разработки приложений для встраиваемых систем и мобильных устройств с ограниченными ресурсами.

•**Области применения:** Устарела, ранее использовалась для мобильных телефонов (не смартфонов), встраиваемых систем (например, банкоматы), но сейчас заменена Android.

3. Виртуальные машины и их роль в JAVA. Архитектура JVM. Основные компоненты: Class Loader, Execution Engine, Garbage Collector.

•**Роль JVM:**

•Абстрагирует аппаратное и программное обеспечение.

•Обеспечивает переносимость Java-приложений на различные платформы.

•Отвечает за выполнение байт-кода.

•Управляет памятью, потоками и безопасностью.

•**Архитектура JVM:**

•**Class Loader:** Загружает классы (.class файлы) в память JVM.

•**Execution Engine:** Выполняет байт-код, интерпретируя его или компилируя в машинный код (JIT).

•**Runtime Data Areas:** Различные области памяти, такие как куча, стек, методная область.

•**Garbage Collector:** Освобождает память, занимаемую неиспользуемыми объектами.

•**Компоненты:**

•**Class Loader:**

•Загружает .class файлы в память.

•Выполняет верификацию байт-кода.

•Ищет и загружает нужные классы из разных мест (например, classpath).

•**Execution Engine:**

•**Интерпретатор:** Читает байт-код и выполняет его построчно.

•**JIT-компилятор:** Динамически компилирует часто используемый байт-код в машинный код для ускорения выполнения.

•**Garbage Collector:**

•Автоматически освобождает память от неиспользуемых объектов.

•Минимизирует утечки памяти.

•Использует различные алгоритмы для сбора мусора.

4. Компиляция Java-программ. Различия между JIT (Just-in-Time) и AOT (Ahead-of-Time) компиляцией. Преимущества и недостатки.

•**Компиляция Java:**

•Исходный код (.java) компилируется в байт-код (.class) с помощью компилятора javac.

•Байт-код выполняется JVM.

•**JIT (Just-in-Time) Компиляция:**

•Компиляция байт-кода в машинный код во время выполнения программы.

•Динамическая оптимизация горячих участков кода.

•**Преимущества:**

•Оптимизация под конкретную платформу и среду выполнения.

•Динамическая адаптация под особенности приложения.

•Переносимость байт-кода.

•**Недостатки:**

•Время на “разогрев” приложения.

•Дополнительная нагрузка на ресурсы во время выполнения.

•**AOT (Ahead-of-Time) Компиляция:**

•Компиляция байт-кода в машинный код до запуска программы.

•**Преимущества:**

•Мгновенный запуск программы.

•Более предсказуемая производительность.

•**Недостатки:**
•Потеря переносимости.
•Не может динамически адаптироваться под среду выполнения.

5. Модель памяти в Java. Основные области памяти JVM: куча (Heap) и стек (Stack), их назначение и различия. Как распределяются объекты и примитивные данные в этих областях? Что такое Young Generation, Old Generation и Metaspace? Как работа сборщика мусора влияет на управление памятью?

•**Основные области памяти:**
•**Куча (Heap):**
•Динамическая область памяти для хранения объектов.
•Общая для всех потоков.
•Управляется сборщиком мусора.
•Объекты выделяются в куче с помощью оператора `new`.
•**Стек (Stack):**
•Область памяти для хранения кадров вызовов методов (локальные переменные, адреса возврата).
•Каждый поток имеет свой собственный стек.
•Локальные примитивные переменные и ссылки на объекты хранятся в стеке.
•**Распределение данных:**
•**Объекты:** Создаются в куче, а ссылка на объект хранится в стеке.
•**Примитивы:** Локальные примитивные переменные хранятся в стеке.
•**Статические переменные:** Хранятся в методологической области (Metaspace).
•**Области кучи:**
•**Young Generation:**
•Область кучи, в которой создаются новые объекты.
•Разделена на Eden Space и Survivor Spaces.
•“Мелкая” уборка мусора происходит часто.
•**Old Generation:**
•Область кучи, в которую перемещаются объекты, пережившие несколько сборок мусора.
•“Большая” уборка мусора происходит реже.
•**Metaspace:**
•Область памяти для хранения метаданных классов, например, статические переменные, константы.
•Заменила PermGen Space в Java 8.
•**Работа сборщика мусора:**

•Отслеживает доступность объектов.
•Освобождает память от недоступных объектов.
•Разделяет память на поколения для более эффективной уборки.
•Различные алгоритмы: Mark-Sweep, Copying, Mark-Compact, и Generational.

6. Основные парадигмы программирования в Java. Объектно-ориентированное, функциональное, многопоточное программирование.

•**Объектно-ориентированное программирование (ООП):**
•**Основные принципы:** Инкапсуляция, Наследование, Полиморфизм, Абстракция.
•Создание классов и объектов, моделирующих реальные сущности.
•Модульность, повторное использование, гибкость.
•Java – один из главных языков ООП.
•**Функциональное программирование (ФП):**
•**Основные принципы:** Функции как объекты первого класса, лямбда-выражения, иммутабельность данных, чистые функции, операции с потоками данных.
•Java частично поддерживает функциональное программирование с Java 8 (лямбды, Stream API).
•Снижает количество побочных эффектов, упрощает параллельное выполнение, улучшает читаемость.
•**Многопоточное программирование:**
•Создание нескольких потоков выполнения в рамках одного процесса.
•Позволяет выполнять несколько задач параллельно.
•Улучшает производительность многоядерных процессоров.
•Java имеет встроенную поддержку многопоточности (классы Thread, Executor).
•Требует осторожности для избежания ошибок синхронизации (race conditions, deadlock).

7. Виртуальные машины и их роль в JAVA. Особенности стандартной HotSpot JVM. GraalVM и другие сторонние виртуальные машины для Java. Основные преимущества и возможности сторонних виртуальных машин.

•**Роль JVM:** (повторение, чтобы закрепить)
•Обеспечивает переносимость Java-кода.
•Выполняет байт-код.
•Управляет памятью и ресурсами.
•**HotSpot JVM:**
•Стандартная JVM от Oracle, является самой распространенной.
•Хорошо оптимизирована для большинства случаев использования.
•Содержит интерпретатор и JIT-компилятор.
•**GraalVM:**

•Сторонняя JVM, разработанная компанией Oracle Labs.
•Позволяет компилировать байт-код в нативный машинный код (AOT) и работает как “универсальная” виртуальная машина, позволяющая выполнять различные языки программирования.

•Преимущества:

•Значительно повышает скорость запуска приложений (AOT).
•Позволяет создавать нативные образы приложений (Native Image).
•Интеграция с другими языками (Python, JavaScript, Ruby, R).

•Недостатки:

•Требуется больше времени для компиляции.
•Менее гибкая в некоторых аспектах, чем HotSpot.

•Другие виртуальные машины:

•OpenJDK: Бесплатная реализация Java, на основе которой разрабатывается HotSpot.
•J9 (Eclipse OpenJ9): Альтернативная виртуальная машина, оптимизированная для работы в облаке.

•Преимущества сторонних JVM:

•Оптимизация под конкретные сценарии (например, GraalVM для AOT, J9 для облака).
•Более гибкое управление памятью.
•Интеграция с другими языками.
•Разные особенности производительности, подходящие под разные случаи.

8. Компиляция и запуск проекта на Java. Обеспечение переносимости кода на различные платформы. Понятие промежуточного байт-кода и его роль в переносимости программ. Чем отличаются методы компиляции JIT (Just-In-Time) и AOT (Ahead-of-Time), и как они влияют на производительность и переносимость?

•Компиляция и запуск:

1.**Написание кода (.java):** Программист пишет исходный код на языке Java.
2.**Компиляция (javac):** Компилятор javac преобразует исходный код в байт-код (.class).
3.**Запуск (java):** Команда java запускает JVM, которая загружает и выполняет байт-код.

•Переносимость:

1.Байт-код выполняется на любой платформе с установленной JVM.
2.Принцип “Write Once, Run Anywhere” (WORA).

•Промежуточный байт-код:

1.Байт-код – это платформонезависимый набор инструкций.

2.JVM интерпретирует или компилирует этот байт-код в машинный код на целевой платформе.
3.Обеспечивает переносимость программы.

- **JIT и AOT (различия):** (повторение)

1. **JIT (Just-in-Time):** Динамическая компиляция байт-кода в машинный код во время выполнения.
- **Производительность:** Динамическая оптимизация, но время на “разогрев”.
- **Переносимость:** Полная переносимость байт-кода.

2. **AOT (Ahead-of-Time):** Компиляция байт-кода в машинный код до запуска программы.
- **Производительность:** Мгновенный запуск, но менее гибкая оптимизация.
- **Переносимость:** Потеря переносимости, бинарник привязан к платформе.

9. Современный инструментарий разработчика Java. Популярные IDE и их возможностей для написания, отладки и сборки кода. Основные системы сборки и их роль в управлении проектами на JAVA. Контроль версий с использованием Git и интеграция с платформами хостинга ИТ-проектов. Использование Docker и Kubernetes для контейнеризации и оркестрации приложений. Инструменты CI/CD для автоматизации сборки, тестирования и деплоя JAVA приложений.

- **IDE (Integrated Development Environment):**
- **IntelliJ IDEA:** Самая популярная, мощная, многофункциональная, платная.
- **Eclipse:** Бесплатная, расширяемая, настраиваемая.
- **Visual Studio Code:** Легкая, гибкая, бесплатная (с плагинами для Java).
- **Возможности:**
  - Автодополнение кода, рефакторинг.
  - Отладка (debugging).
  - Управление проектами.
  - Интеграция с системами контроля версий, сборки, тестирования.
  - Редактирование и навигация по коду.
- **Системы сборки:**
- **Maven:** Популярен для Java EE, основан на XML, декларативный.
- **Gradle:** Более гибкий, использует Groovy или Kotlin DSL, подходит для больших проектов.
- **Роль:**
  - Управление зависимостями (библиотеки).
  - Сборка проекта в JAR или WAR.
  - Запуск тестов.
  - Автоматизация процессов сборки.

•**Контроль версий (Git):**
•Распределенная система контроля версий.
•Отслеживание изменений в коде, возможность отката.
•Работа в команде.
•**Платформы:** GitHub, GitLab, Bitbucket.
•**Интеграция:** Работа из IDE, pull requests, code reviews.
•**Контейнеризация (Docker):**
•Упаковка приложений и их зависимостей в контейнеры.
•Изоляция приложений друг от друга.
•Простота развертывания.
•**Оркестрация (Kubernetes):**
•Управление контейнерами на кластере серверов.
•Масштабирование, отказоустойчивость.
•Автоматизация развертывания.
•**CI/CD (Continuous Integration / Continuous Delivery):**
•**CI:** Автоматическая сборка и тестирование кода при каждом изменении.
•**CD:** Автоматическая доставка и развертывание приложения на целевой среде.
•**Инструменты:** Jenkins, GitLab CI, GitHub Actions, Travis CI.
•**Цель:** Сокращение времени от написания кода до его развертывания.

10. Современные фреймворки для разработки на Java. Особенности Spring Framework. Основные возможности Hibernate. Основные причины использования данных фреймворков при разработке на JAVA.

•**Spring Framework:**
•**Особенности:**
•Широко используется для разработки корпоративных приложений, микросервисов, веб-приложений, backend-систем.
•Модульная архитектура (Spring MVC, Spring Boot, Spring Security, Spring Data JPA).
•Поддержка Dependency Injection (DI) и Aspect-Oriented Programming (AOP).
•Большое сообщество и множество библиотек.
•**Причины использования:**
•Упрощает разработку, сокращает количество повторяющегося кода.
•Повышает качество и надежность приложений.
•Позволяет легко интегрироваться с другими технологиями.
•Быстрое прототипирование и разработка.
•**Hibernate:**
•**Особенности:**
•ORM (Object-Relational Mapping) фреймворк.
•Позволяет работать с базами данных, не используя напрямую SQL.

•Сопоставляет объекты Java с таблицами в базе данных.
•Имеет множество функций для работы с данными: запросы, транзакции, кеширование.
•**Причины использования:**
•Упрощает работу с базами данных.
•Снижает вероятность ошибок, связанных с SQL.
•Обеспечивает независимость от конкретной СУБД.
•Ускоряет разработку.
•Обеспечивает объектно-ориентированный подход к базам данных.
•**Причины использования фреймворков:**
•Ускорение разработки (готовые компоненты и шаблоны).
•Упрощение кода, уменьшение повторяющегося кода.
•Снижение количества ошибок (тестированные компоненты).
•Повышение производительности и надежности приложений.
•Стандартизация подхода к разработке.

11. Объектная модель Java. Основные принципы объектной модели в Java: классы, объекты, интерфейсы, наследование и инкапсуляция. Класс Object и методы, которые он предоставляет.

•**Объектная модель Java:**
•**Классы:** Шаблоны для создания объектов, определяют свойства и поведение.
•**Объекты:** Экземпляры классов, конкретные сущности в программе.
•**Интерфейсы:** Контракты, определяющие методы, которые должен реализовать класс.
•**Наследование:** Создание новых классов на основе существующих, переиспользование кода.
•**Инкапсуляция:** Сокрытие внутреннего состояния объекта и предоставление доступа к нему через методы.
•**Класс Object:**
•Базовый класс для всех классов в Java.
•**Методы:**
•`toString()`: Возвращает строковое представление объекта.
•`equals(Object obj)`: Сравнивает объекты на равенство.
•`hashCode()`: Возвращает хеш-код объекта.
•`getClass()`: Возвращает класс объекта.
•`clone()`: Создает копию объекта.
•`finalize()`: Метод вызывается перед удалением объекта сборщиком мусора (устарел).

•`notify(), notifyAll(), wait()`: Методы для работы с потоками.

12. Пакеты в Java. Основное предназначение. Структура, организация, использование в программировании (импорт пакетов).

•**Пакеты:**
•Способ организации классов и интерфейсов в Java.
•Предотвращают конфликты имен.
•Упрощают управление кодом.
•Обеспечивают доступность/видимость классов и методов.
•**Предназначение:**
•Логическое группирование классов и интерфейсов.
•Разделение пространства имен (избежание коллизий).
•Управление уровнем доступа (public, protected, default (package-private)).
•**Структура:**
•Иерархическая структура (папки на файловой системе).
•Имена пакетов пишутся с маленькой буквы (обычно домен в обратном порядке, например `com.example.myapp`).
•Например, `java.util` или `javax.swing`.
•**Использование:**
•**Объявление пакета:** `package com.example.myapp;` (в начале .java файла).
•**Импорт пакетов:** `import java.util.ArrayList;` (импорт конкретного класса) или `import java.util.*;` (импорт всех классов из пакета).
•**Полное имя класса:** `java.lang.String` (если класс не импортирован).

13. Синтаксис и лексика Java. Основные элементы лексики языка: ключевые слова, идентификаторы, литералы, комментарии, операторы и разделители. Правила именования идентификаторов. Соглашения по оформлению кода.

•**Лексические элементы:**
•**Ключевые слова:** Зарезервированные слова, имеющие специальное значение (например, `class`, `public`, `static`, `if`, `for`, `int`, `void`).
•**Идентификаторы:** Имена переменных, классов, методов, пакетов, созданные программистом.
•**Литералы:** Представления конкретных значений (числа, строки, символы, boolean).
•**Комментарии:** Заметки в коде, игнорируемые компилятором.
•`// однострочный комментарий`
•`/* многострочный комментарий */`
•`/** javadoc комментарий */`

•**Операторы:** Символы для выполнения операций (арифметические, логические, сравнения, присваивания, тернарные).
•**Разделители:** Символы для разделения элементов синтаксиса (круглые скобки `()`, фигурные скобки `{}`, квадратные скобки `[]`, точка `.`, точка с запятой `;`, запятая `,`, пробелы).
•**Правила именования идентификаторов:**
•Начинаются с буквы, знака доллара `$`, или символа подчеркивания `_`.
•Состоят из букв, цифр, знаков доллара и символов подчеркивания.
•Регистрозависимые.
•Не могут быть ключевыми словами.
•**Соглашения по оформлению кода (Code Conventions):**
•**Имена классов:** Начинаются с заглавной буквы (PascalCase).
•**Имена методов и переменных:** Начинаются с маленькой буквы (camelCase).
•**Имена констант:** Все буквы заглавные (UPPER_CASE).
•**Отступы:** Использовать 4 пробела или табуляцию.
•**Комментарии:** Должны быть краткими и понятными.
•**Пробелы:** Использовать для читаемости (например, вокруг операторов).
•**Длина строк:** Не превышать 80-120 символов.

14. Типы данных в Java. Примитивные типы данных, объявление и присваивание переменных. Отличия примитивных типов данных от ссылочных.

•**Примитивные типы данных:**
•`byte`: 8-битное целое число.
•`short`: 16-битное целое число.
•`int`: 32-битное целое число.
•`long`: 64-битное целое число.
•`float`: 32-битное число с плавающей точкой.
•`double`: 64-битное число с плавающей точкой.
•`char`: 16-битный символ Unicode.
•`boolean`: Логическое значение (true или false).
•**Объявление и присваивание переменных:**

```
int age; // Объявление переменной типа int с именем age.
age = 30; // Присваивание значения 30 переменной age.

double price = 19.99; // Объявление и присваивание в одной строке.
char initial = 'J';
boolean isAdult = true;
```

•**Отличия от ссылочных типов:**

•**Хранение данных:** Примитивные типы хранят сами значения в памяти (в стеке).

•**Размер в памяти:** Размер предопределен (например, int всегда 4 байта).

•**Нулевое значение:** У примитивных типов есть значения по умолчанию, для числовых типов это 0, для boolean это false, для char это '\u0000'.

•**Нельзя вызывать методы:** У примитивных типов нет методов.

•**Сравнение:** Сравниваются по значениям (например, `a == b`).

15. Типы данных в Java. Ссылочные типы данных, объявление и присваивание переменных. Отличия ссылочных типов данных от примитивных. Роль классов-оберток (Wrapper Classes) для работы с примитивами.

•**Ссылочные типы данных:**

•Классы, массивы, интерфейсы, перечисления.

•Хранят ссылку (адрес) на объект, расположенный в куче (heap).

•**Объявление и присваивание переменных:**

```
String name; // Объявление переменной типа String.
name = "John Doe"; // Присваивание ссылки на объект-строку.

Integer count = new Integer(10); // Integer - ссылочный тип.
int[] numbers = new int[5]; // Объявление массива.
```

•**Отличия от примитивных типов:**

•**Хранение данных:** Хранят ссылки на объекты в куче (heap), а сами переменные находятся в стеке.

•**Размер в памяти:** Размер зависит от объекта.

•**Нулевое значение:** Значение по умолчанию `null` (ссылка никуда не ведет).

•**Методы:** У ссылочных типов есть методы (например, `name.length()`).

•**Сравнение:** По умолчанию сравниваются ссылки (например, `obj1 == obj2`), а не содержимое. Для сравнения содержимого нужно использовать метод `equals()`.

•**Классы-обертки (Wrapper Classes):**

•`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`.

•Классы-обертки позволяют использовать примитивные типы как объекты.

•**Роль:**

•Предоставляют методы для работы с примитивными типами.

•Необходимы для использования в коллекциях (например, `ArrayList<Integer>`).

•Могут быть `null` (в отличие от примитивов).

•Автоматическая упаковка (autoboxing) и распаковка (unboxing) между примитивами и обертками.

```
Integer num = 10; // Автоупаковка (autoboxing) int -> Integer.
```

```
int value = num; // Автораспаковка (unboxing) Integer -> int.
```

16. Константы в Java. Понятие констант и их объявление с использованием ключевого слова final. Основные правила и соглашения по именованию констант. Примеры создания констант для примитивных типов данных и строк. Как константы помогают обеспечить неизменность данных и улучшают читаемость кода?

•**Константы:** Переменные, значение которых не может быть изменено после инициализации.

•**Ключевое слово `final`:**

•Используется для объявления констант.

•`final` переменные необходимо инициализировать при объявлении.

```
final int MAX_VALUE = 100;
final double PI = 3.14159;
final String APPLICATION_NAME = "My App";
```

•**Правила и соглашения по именованию констант:**

•Все буквы заглавные.

•Слова разделяются символом подчеркивания `_`.

•Имена должны быть описательными.

•**Примеры:**

```
final int MAX_THREADS = 8;
final double GRAVITY = 9.81;
final String FILE_EXTENSION = ".txt";
final boolean DEBUG_MODE = true;
```

•**Польза констант:**

•**Неизменность данных:** Защита от случайного изменения значений.

•**Читаемость кода:** Описательные имена облегчают понимание кода.

•**Простота обслуживания:** Изменение константы в одном месте влияет на весь код.

17. Ключевое слово `var` в Java. Особенности использования `var` для объявления локальных переменных. Как происходит неявное выведение типа переменной компилятором? Ограничения на использование `var`: недопустимость для полей класса, параметров методов и возвращаемых типов.

•**Ключевое слово `var` (Java 10+):**

•Используется для объявления локальных переменных (внутри методов).

•Компилятор автоматически определяет тип переменной на основе присваиваемого значения (неявное выведение типа).

```
var message = "Hello, World!"; // Тип message - String
```

```
var count = 10; // Тип count - int
var price = 19.99; // Тип price - double
```

- **Неявное выведение типа:**
- Компилятор анализирует выражение справа от знака `=`.
- Тип переменной становится типом присвоенного значения.
- **Ограничения:**
- `var` нельзя использовать для:
- Полей класса (переменных экземпляра или статических).
- Параметров методов.
- Возвращаемых типов методов.
- Нельзя объявить переменную без инициализации (`var x;` не скомпилируется).
- `var` не может быть `null` (т.е. нельзя `var x = null`).
- **Когда использовать `var`:**
- Когда тип переменной очевиден из контекста.
- Для сокращения кода и повышения читаемости (особенно с длинными типами).
- Для избежания дублирования типа.

18. Соглашения по оформлению кода Java. Java Code Conventions и её значение для совместной работы.

- **Java Code Conventions:**
- Стандартный набор правил и соглашений по форматированию кода на Java.
- Официальная версия от Oracle (но могут быть и другие).
- **Основные правила:**
- Именование (классы, методы, переменные, константы).
- Отступы и форматирование (4 пробела, длина строк).
- Комментарии (Javadocs, однострочные, многострочные).
- Пустые строки (для разделения логических блоков).
- Организация import-выражений.
- Скобки (где и как ставить).
- и др.
- **Значение для совместной работы:**
- **Единообразие стиля:** Все разработчики пишут код в одном стиле.
- **Читаемость кода:** Легче понять код, написанный другими.
- **Сопровождаемость:** Проще поддерживать код.
- **Экономия времени:** Разработчики не тратят время на споры о стиле.
- **Снижение ошибок:** Единый стандарт упрощает чтение и понимание кода, снижает вероятность ошибок.

•**Инструменты:** IDE и статические анализаторы помогают автоматически проверять соответствие Code Conventions.

19. Класс и экземпляры класса. Что такое класс в Java и как происходит создание объектов (инстанцирование) с использованием ключевого слова new? Примеры создания и использования экземпляров класса.

•**Класс:**

•Чертеж, шаблон для создания объектов.

•Определяет свойства (поля) и поведение (методы) объектов.

•Состоит из полей, конструкторов, методов и вложенных классов (если есть).

•**Инстанцирование (создание объектов):**

•Создание экземпляра класса с использованием ключевого слова new.

•Объект получает память в куче.

•Конструктор класса вызывается для инициализации объекта.

•**Примеры:**

```
// Класс
class Dog {
    String name;
    int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void bark() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Создание объектов (экземпляров) класса Dog
        Dog myDog = new Dog("Buddy", 3); // Вызывается конструктор Dog
        Dog anotherDog = new Dog("Bella", 5);

        // Использование объектов
        System.out.println(myDog.name); // Вывод: Buddy
        myDog.bark(); // Вывод: Woof!

        System.out.println(anotherDog.name); // Вывод: Bella
    }
}
```

20. Записи (Records) в Java. Какие возможности они предоставляют и в чем их отличие от обычных классов? Примеры использования записей.

•**Записи (Records) (Java 14+):**

•Специальный тип класса, предназначенный для представления неизменяемых данных.

•Автоматически генерируют конструктор, геттеры, `equals()`, `hashCode()`, и `toString()`.

•Упрощают создание классов, предназначенных для хранения данных.

•**Отличия от обычных классов:**

•**Иммутабельность:** Поля записи являются `final` (их нельзя изменить после создания).

•**Меньше кода:** Автоматически генерируется значительная часть кода.

•**Предназначение:** В основном для представления данных, а не поведения.

•Не могут иметь объявленные переменные экземпляра (только компоненты).

•Не могут наследовать от других классов, но могут реализовывать интерфейсы.

•**Примеры:**

```
record Point(int x, int y) {}

record Book(String title, String author, int year) {}

public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(10, 20);
        System.out.println(p1.x()); // Вывод: 10 (автоматический геттер)
        System.out.println(p1); // Вывод: Point[x=10, y=20] (автоматический toString)

        Book book1 = new Book("Java for Beginners", "John Doe", 2023);

        System.out.println(book1.title()); // Вывод: Java for Beginners
        System.out.println(book1); // Вывод: Book[title=Java for Beginners, author=John Doe, year=2023]
    }
}
```

21. Запечатанные (Sealed) классы. Как они ограничивают наследование и для чего используются?

•**Запечатанные классы (Sealed Classes) (Java 17+):**

•Ограничивают наследование только теми классами, которые явно объявлены как разрешенные.

•Управляют иерархией классов.

•Повышают предсказуемость и безопасность кода.

•**Ограничения наследования:**

•Используется ключевое слово `sealed`.

•Перечисляются разрешенные классы, которые могут наследовать (`permits`).

•Не разрешенные классы не могут наследовать от запечатанного класса.

•Могут быть интерфейсы `sealed` (с применением `permits`).

- **Использование:**
- Моделирование ограниченной иерархии классов.
- Создание более контролируемых API.
- Улучшение читаемости и сопровождаемости кода.
- **Примеры:**

```
sealed class Shape permits Circle, Rectangle, Triangle {}

final class Circle extends Shape {}

non-sealed class Rectangle extends Shape {} // non-sealed класс может быть расширен

sealed interface  User permits Customer, Admin {

}

 record Customer (String name, String email) implements User {
}

record Admin (String name) implements User{
}

 // Ошибка - класс Square не объявлен в permits
// class Square extends Shape {}
```

22. Инкапсуляция в Java. Понятие инкапсуляции как механизма защиты данных и управления доступом к ним. Реализация инкапсуляции с использованием модификаторов доступа (private, protected, public, package-private). Роль геттеров и сеттеров в обеспечении контроля за изменением данных объекта. Примеры нарушения инкапсуляции и способы предотвращения этих ошибок.

- **Инкапсуляция:**
- Сокрытие внутреннего состояния объекта (данных) от внешнего мира.
- Предоставление доступа к данным через методы (геттеры и сеттеры).
- Защита данных от несанкционированного доступа и изменения.
- Обеспечивает модульность и гибкость.
- **Модификаторы доступа:**
- `private`: Доступ только внутри класса.
- `protected`: Доступ внутри пакета и в подклассах (даже из других пакетов).
- `public`: Доступ из любого места.
- `package-private` (default, нет модификатора): Доступ внутри пакета.
- **Геттеры и сеттеры:**
- Геттеры (`get...`) : методы для получения значения полей.
- Сеттеры (`set...`) : методы для установки значений полей.

•Позволяют контролировать доступ к полям и вводить дополнительную логику при чтении и изменении данных.

•Реализация инкапсуляции:

```
class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
         if (name == null || name.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be null or empty");
         }
        this.name = name;
    }
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
        this.age = age;
    }
}

 public static void main(String[] args) {
        Person person = new Person();
        person.setName("John Doe");
        person.setAge(30);
        System.out.println("Name " + person.getName() + " age " +
person.getAge());

         try {
        person.setAge(-5); // Нарушение инкапсуляции - некорректный возраст
    } catch (IllegalArgumentException e) {
          System.out.println("Error: " + e.getMessage()); // Обработка ошибки с
некорректным возрастом.
    }

        System.out.println("Name " + person.getName() + " age " +
person.getAge());
    }
```

•Нарушение инкапсуляции:

•Прямой доступ к `private` полям (не через геттеры/сеттеры).

•Неправильная валидация данных в сеттерах.

•Предотвращение:

•Использовать `private` для полей.

•Предоставлять доступ через `public` геттеры и сеттеры.

•Выполнять валидацию данных в сеттерах.

•Избегать `public` полей.

23. Модификаторы доступа. Какие уровни доступа существуют в Java? Как модификаторы доступа используются для контроля видимости классов, полей и методов?

•Уровни доступа:

•**`private`:** Доступ только внутри класса.

•**`protected`:** Доступ внутри пакета и в подклассах (даже из других пакетов).

•**`public`:** Доступ из любого места (из любого пакета, из любого класса).

•**`package-private` (default):** Доступ только внутри пакета (если модификатор не указан).

•Контроль видимости:

•Классы:

•`public`: Виден из любого пакета.

•`package-private`: Виден только внутри пакета.

•Поля и методы:

•`private`: Видны только внутри класса.

•`protected`: Видны внутри пакета и в подклассах.

•`public`: Видны из любого места.

•`package-private`: Видны только внутри пакета.

•Таблица видимости:

Модификатор	Внутри класса	Внутри пакета	В подклассах	Везде
`private`	✅	❌	❌	❌
`package-private`	✅	✅	❌	❌
`protected`	✅	✅	✅	❌
`public`	✅	✅	✅	✅

24. Модификатор `final`. Применение `final` к переменным, методам и классам. Как он предотвращает изменения данных, поведение методов и наследование?

•Модификатор `final`:

•Означает “окончательный” или “неизменяемый”.

•Применение к переменным:

•Переменная становится константой.

•Значение можно присвоить только один раз (при объявлении или в конструкторе).

```
final int MAX_VALUE = 100;
final String NAME; // Можно инициализировать в конструкторе
public MyClass(String name){
    this.NAME = name;
}
```

•Применение к методам:

•Метод нельзя переопределить в подклассах (метод становится не переопределяемым).

```
class Parent {
    final void myMethod() { /*...*/ }
}

class Child extends Parent {
    // Ошибка компиляции: нельзя переопределить final метод
    //  void myMethod(){ /* ... */ }
}
```

•Применение к классам:

•Класс нельзя наследовать (класс становится не наследуемым).

```
final class FinalClass {
    // ...
}

    // Ошибка компиляции: нельзя наследовать от final класса
    // class SubFinalClass extends FinalClass{ /*...*/ }
```

•Предотвращение:

•Данных (переменные): Предотвращает изменения значения.

•Поведения (методы): Предотвращает переопределение метода.

•Наследования (классы): Предотвращает создание подклассов.

•Используется для:

•Создания констант.

•Защиты метода от изменения логики в подклассах.

•Гарантии того, что класс не может быть расширен.

25. Конструкторы в Java. Понятие конструктора и его роль в создании объектов. Различия между конструктором и методом. Типы конструкторов. Как реализовать перегрузку конструкторов?

•Конструктор:

•Специальный метод, который вызывается при создании объекта класса.

•Используется для инициализации полей объекта.

•Имя конструктора совпадает с именем класса.

•Не имеет возвращаемого значения (даже `void` не указывается).

•**Различия между конструктором и методом:**

•**Назначение:** Конструктор - для инициализации объекта; метод - для выполнения действий.

•**Имя:** Имя конструктора совпадает с именем класса; имя метода может быть любым.

•**Возвращаемое значение:** Конструктор не имеет возвращаемого значения; метод может возвращать значение.

•**Вызов:** Конструктор вызывается с помощью `new`; метод вызывается через имя объекта.

•**Типы конструкторов:**

•**Конструктор по умолчанию:**

•Создается компилятором автоматически, если в классе нет ни одного конструктора.

•Не принимает параметров.

•Инициализирует поля значениями по умолчанию.

•**Конструктор с параметрами:**

•Создается программистом.

•Принимает параметры для инициализации полей.

•**Перегрузка конструкторов:**

•Создание нескольких конструкторов с разными списками параметров в одном классе (как и с методами).

•Позволяет создавать объекты с разными способами инициализации.

```
class Person {
    String name;
    int age;

    // Конструктор по умолчанию
    public Person() {}

    // Конструктор с одним параметром
    public Person(String name) {
        this.name = name;
    }

    // Конструктор с двумя параметрами
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

    public static void main(String[] args) {
    Person person1 = new Person();
    Person person2 = new Person("John");
    Person person3 = new Person("John", 30);
```

```
}
```

26. Конструкторы в Java. Понятие конструктора и его роль в создании объектов. Использование ключевого слова `this` для вызова одного конструктора из другого. Особенности работы конструкторов в наследовании, вызов конструктора родительского класса через `super`.

•**Конструкторы:** (повторение)
•Используются для создания и инициализации объектов.
•Название конструктора совпадает с названием класса.
•Не имеют возвращаемого значения.
•**`this()`:**
•Вызывает один конструктор этого же класса из другого конструктора.
•Должен быть первым оператором в конструкторе.
•Используется для переиспользования логики из других конструкторов.

```
class Product {
    String name;
    double price;

    public Product(String name) {
        this.name = name;
    }

    public Product(String name, double price) {
        this(name);  // Вызов конструктора Product(String name)
        this.price = price;
    }
}
```

•**Конструкторы в наследовании:**
•При создании объекта подкласса сначала вызывается конструктор родительского класса.
•Конструкторы не наследуются.
•**`super()`:**
•Вызывает конструктор родительского класса из конструктора подкласса.
•Должен быть первым оператором в конструкторе подкласса.
•Используется для инициализации полей, унаследованных от родителя.
•Если `super()` не указан явно, то вызывается конструктор родительского класса без параметров (по умолчанию).

```
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }
}
```

```
class Dog extends Animal {
    String breed;

    public Dog(String name, String breed) {
        super(name); // Вызов конструктора Animal(String name)
        this.breed = breed;
    }
}
```

27. Блоки инициализации. Виды блоков инициализации: статические и нестатические. Их роль в подготовке объектов и классов. Примеры использования блоков для сокращения повторяющегося кода.

- **Блоки инициализации:**
- Блоки кода, которые выполняются при создании объекта или при загрузке класса.
- Используются для инициализации полей класса.
- **Виды:**
- **Нестатические блоки инициализации:**
- Выполняются при создании каждого объекта (экземпляра) класса.
- Могут обращаться к нестатическим полям.
- Используются для инициализации общих для всех объектов данных.
- **Статические блоки инициализации:**
- Выполняются один раз при загрузке класса (перед созданием первого объекта).
- Могут обращаться только к статическим полям.
- Используются для инициализации статических полей класса.
- **Роль:**
- **Подготовка объектов:** Инициализация полей при создании объектов.
- **Подготовка классов:** Инициализация статических полей при загрузке класса.
- **Сокращение повторяющегося кода:**
- Используются для общей логики инициализации, чтобы не дублировать код в разных конструкторах.
- **Примеры:**

```
class MyClass {
    int id;
    String name;

    // Нестатический блок инициализации
    {
        id = generateId();
        System.out.println("Non-static initializer block executed");
    }

    // Статический блок инициализации
    static {
        System.out.println("Static initializer block executed");
```

```
    }

    public MyClass(String name) {
        this.name = name;
         System.out.println("Constructor with name param");
    }

     public MyClass() {
          System.out.println("Default constructor");
       }

    private int generateId() {
      return (int) (Math.random()*1000);
    }
}

 public static void main(String[] args) {
    MyClass obj1 = new MyClass("Object 1");
     System.out.println("Object 1 id: " + obj1.id);
    MyClass obj2 = new MyClass();
    System.out.println("Object 2 id: " + obj2.id);
 }
```

28. Статические блоки инициализации. Примеры и использование статических блоков для выполнения кода при загрузке класса. Их роль в инициализации общих данных.

- **Статические блоки инициализации:**
- Выполняются только один раз при загрузке класса в память JVM.
- Выполняются до создания любого объекта этого класса.
- Могут обращаться только к статическим полям класса.
- Используются для инициализации статических полей, выполнения настроек при загрузке класса.
- **Роль:**
- **Инициализация общих данных:** Статические блоки используются для инициализации статических полей, которые являются общими для всех объектов этого класса.
- **Выполнение кода при загрузке:** Выполнение кода, который должен быть выполнен один раз при загрузке класса (например, загрузка конфигурации, создание подключения к БД).
- **Примеры:**

```
class MySettings {

    static String dbUrl;
    static String api_key;
    static final String DEFAULT_LANG;

    static {
        dbUrl = "jdbc:mysql://localhost:3306/mydb";
        api_key = generateKey();
```

```
        DEFAULT_LANG = "EN";
        System.out.println("Static initialization block executed");
    }

    public static String getApiKey(){
        return api_key;
    }

    private static String generateKey(){
        return String.valueOf((int)(Math.random() * 100000));
    }
}

public static void main(String[] args) {
    System.out.println("Database url: " + MySettings.dbUrl);
    System.out.println("Api key " + MySettings.getApiKey());
}
```

29. Модификатор `static`. Особенности использования `static` для полей, методов и блоков. Различия между статическими и нестатическими членами класса. Примеры применения для создания общих ресурсов.

- **Модификатор `static`:**
- Относит поля, методы и блоки к классу, а не к объекту.
- Статические члены (поля, методы) доступны напрямую через имя класса, без создания объекта.
- **Статические поля:**
- Общие для всех объектов класса (все объекты имеют доступ к одному и тому же значению).
- Инициализируются при загрузке класса.
- **Статические методы:**
- Могут вызываться напрямую через имя класса.
- Не могут обращаться к нестатическим полям или методам (только к статическим).
- Используются для операций, связанных с классом (а не конкретным объектом).
- **Статические блоки инициализации:**
- Выполняются один раз при загрузке класса.
- Используются для инициализации статических полей.
- **Различия:**
- **Статические члены:** Принадлежат классу.
- **Нестатические члены:** Принадлежат объекту.
- Статические члены доступны через имя класса, нестатические - через объект.
- **Примеры применения (общие ресурсы):**

```
class Counter {
    static int count = 0;

    public Counter() {
        count++;
```

```
    }

    public static int getCount() {
        return count;
    }
}

 public static void main(String[] args) {
        Counter obj1 = new Counter();
        Counter obj2 = new Counter();
        Counter obj3 = new Counter();
        System.out.println("Count: " + Counter.getCount()); // Вывод: 3
 }


class MathUtils {
      static double PI = 3.14159;

    public static double calculateArea(double radius){
        return PI * radius * radius;
    }
 }

  public static void main(String[] args) {
      double area = MathUtils.calculateArea(5); // Вызов статического метода
через класс MathUtils
        System.out.println("Area: " + area);
    }
```

30. Ключевое слово `this`. Использование `this` для доступа к полям и методам объекта, вызова других конструкторов и передачи текущего объекта. Примеры решения конфликтов имен с помощью `this`.

- **Ключевое слово `this`:**
- Ссылка на текущий объект (экземпляр класса).
- Используется для доступа к полям и методам объекта внутри этого объекта.
- **Использование:**
- **Доступ к полям:** Если имя локальной переменной совпадает с именем поля, `this` используется для доступа к полю объекта.

```
 class Point {
      int x;
      int y;

      public Point(int x, int y) {
          this.x = x; // this.x - поле, x - локальная переменная
          this.y = y;
    }

    public void printCoordinates(){
        System.out.println("Coordinates x=" + this.x + " y=" + this.y);
    }
}
```

•**Доступ к методам:** `this` используется для вызова методов текущего объекта. (Чаще всего используется без this, не явно, но может быть полезно, например, для передачи текущего объекта в качестве параметра)

```
class MyClass {
        void method1() {
            System.out.println("Method 1 called");
            this.method2(); // Вызов метода method2 текущего объекта
        }
       void method2(){
            System.out.println("Method 2 called");
        }
    }
```

•**Вызов других конструкторов:** `this()` используется для вызова одного конструктора этого же класса из другого (как обсуждали ранее).

•**Передача текущего объекта:** `this` может использоваться для передачи текущего объекта в качестве параметра в метод.

```
class SomeClass{

    void process(SomeClass obj) {
        System.out.println("Processing object ...");
    }
    void execute(){
        process(this); // передача текущего объекта
    }
}
```

•**Конфликты имен:**

•`this` разрешает конфликт имен между локальной переменной и полем объекта.

31. Концепция неизменяемых классов. Что делает класс неизменяемым? Использование `final` для предотвращения изменений. Примеры создания неизменяемых объектов.

•**Неизменяемый класс (Immutable Class):**

•Объекты класса не могут быть изменены после создания.

•Состояние объекта остается постоянным.

•**Принципы неизменяемости:**

•**Все поля должны быть `private final`.**

•**Нет сеттеров (методов для изменения полей).**

•**Конструктор должен инициализировать все поля.**

•**Если поле является ссылочным типом, то должна создаваться копия объекта**.

•**Класс должен быть `final` (нельзя наследовать).** (Это необязательно, но обычно используется, т.к. подкласс может нарушить неизменность)

•Геттеры должны возвращать копии изменяемых объектов (чтобы пользователь не смог изменить состояние объекта по ссылке).

•**Польза:**

•Безопасность в многопоточной среде (нет race condition).

•Проще понимать и отлаживать.

•Может использоваться как ключ в `HashMap`.

•Снижает сложность кода.

•**Примеры:**

```
final class ImmutablePoint {
    private final int x;
    private final int y;

    public ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}

 final class ImmutableList {
      private final List<String> list;

     public ImmutableList(List<String> list) {
          this.list = new ArrayList<>(list); //создание копии чтобы состояние не
было изменено из вне
      }

      public List<String> getList(){
          return new ArrayList<>(list); // возврат копии
    }
}

  public static void main(String[] args) {
        ImmutablePoint p1 = new ImmutablePoint(10, 20);
     System.out.println("X=" + p1.getX() + " Y=" + p1.getY());
    }
```

32. Создание объектов. Отличие фабричных методов от стандартного создания объектов с использованием `new`. Примеры использования фабричных методов.

•**Фабричный метод:**

•`static` метод класса, который возвращает объект этого класса.

•Инкапсулирует логику создания объектов.

•Альтернатива использованию `new`.

•**Отличие от `new`:**

•`new`: Непосредственное создание объекта, явное использование конструктора.
•Фабричный метод: Гибкость, контроль, возможность кэширования, возвращение подтипов.
•**Преимущества:**
•**Гибкость:** Можно контролировать процесс создания объекта, например, кэшировать объекты или возвращать подклассы.
•**Инкапсуляция:** Скрывает логику создания объекта от клиента.
•**Удобство:** Более читаемый код, можно дать методу осмысленное имя (например, `createFrom`, `valueOf`).
•**Возможность возвращать подклассы:**
•Фабричный метод может возвращать объекты различных подклассов, что не возможно при использовании `new`
•**Примеры:**

```
class User {
    private String username;
    private String email;

    private User(String username, String email) {
        this.username = username;
         this.email = email;
    }

     // Фабричный метод
    public static User createUser(String username, String email) {
        if (username == null || username.isEmpty()) {
            throw new IllegalArgumentException("Username cannot be empty");
        }
       if (email == null || email.isEmpty()) {
            throw new IllegalArgumentException("Email cannot be empty");
        }

        return new User(username, email);
    }

    public String getUsername(){
        return username;
    }
}

public static void main(String[] args) {
       User user1 =  User.createUser("test_user", "test@email.com");
        System.out.println("User name: " + user1.getUsername());
       // Ошибка нельзя создавать через конструктор так как он private
      // User user2 = new User("user2", "email2@mail.com");
    }

interface Shape {
      void draw();
}

class Circle implements Shape {
```

```
    @Override
     public void draw(){
        System.out.println("Drawing a circle");
    }
}
class Square implements Shape {
```

(); } else if (shapeType.equalsIgnoreCase(“square”)){ return new Square(); } else { throw new IllegalArgumentException(“Invalid shape”); } } }

```
 public static void main(String[] args) {
         Shape shape1 = ShapeFactory.createShape("circle");
        shape1.draw(); //вывод: Drawing a circle
        Shape shape2 = ShapeFactory.createShape("square");
        shape2.draw(); //вывод: Drawing a square
    }
```

33. Рефлексия в Java. Возможности рефлексии для создания объектов и вызова методов во время выполнения. Примеры использования рефлексии для создания объектов.

•Рефлексия (Reflection):

•Возможность исследовать и изменять структуру класса (и его объектов) во время выполнения программы.

•Получение информации о классах, полях, методах, конструкторах.

•Динамическое создание объектов и вызов методов.

•Возможности:

•Получение информации о классе: имя, поля, методы, конструкторы.

•Создание объектов через конструкторы класса.

•Вызов методов объекта.

•Изменение значений полей объекта.

•Примеры создания объектов:

```
class Person {
    private String name;
    private int age;

    public Person() {
         this("default name", 0);
     }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void printInfo() {
         System.out.println("Name: " + name + ", Age: " + age);
    }
}

 public static void main(String[] args) throws Exception {
```

```
        // Получение объекта Class
        Class<?> personClass = Class.forName("Person");

        // Создание объекта с помощью конструктора без параметров
         Person person1 = (Person)
personClass.getDeclaredConstructor().newInstance();
         person1.printInfo(); //Name: default name, Age: 0
        // Создание объекта с помощью конструктора с параметрами
        Person person2 = (Person)
personClass.getDeclaredConstructor(String.class, int.class)
        .newInstance("John Doe", 30);
         person2.printInfo(); // Name: John Doe, Age: 30
        // Получение методов класса
        Method method = personClass.getMethod("printInfo");
        // Вызов метода на объекте
           method.invoke(person1); //Name: default name, Age: 0
    }
```

•Когда использовать:

•Фреймворки (Dependency Injection, ORM).

•Сериализация/десериализация.

•Динамическая загрузка классов.

•Инструменты тестирования и отладки.

•Когда необходимо работать с классами, о которых неизвестно во время компиляции.

•Предупреждение:

•Рефлексия медленнее, чем обычный вызов методов.

•Может ухудшать безопасность кода.

•Сложно читать и сопровождать код с рефлексией.

34. Жизненный цикл объектов в JAVA. Роль сборщика мусора в управлении памятью. Примеры оптимизации работы объектов в Java.

•Жизненный цикл объекта:

1.**Создание:** Объект создается с помощью `new`.

2.**Использование:** Объект используется в программе.

3.**Удаление:** Когда объект больше не нужен, сборщик мусора (Garbage Collector) освобождает память, занимаемую объектом.

•Сборщик мусора (Garbage Collector):

1.Автоматически освобождает память, занятую неиспользуемыми объектами.

2.Работает в фоновом режиме.

3.Отслеживает доступность объектов по ссылкам.

4.Предотвращает утечки памяти.

5.Использует различные алгоритмы (Mark-Sweep, Copying, Mark-Compact, Generational GC).

•Оптимизация работы объектов:

1.**Переиспользование объектов:** Вместо создания новых объектов, использовать уже существующие (например, StringBuilder вместо String для операций с текстом).

```
// Плохо (создается много объектов String)
    String result = "";
    for (int i = 0; i < 10000; i++) {
        result += i + ",";
    }

// Хорошо (используется StringBuilder)
    StringBuilder result2 = new StringBuilder();
    for(int i = 0; i < 10000; i++){
        result2.append(i).append(",");
    }
```

2.**Освобождение ресурсов:** Закрывать открытые ресурсы (файлы, потоки, соединения) в блоке `finally`.

3.**Кэширование:** Использовать кэши для часто используемых объектов.

4.**Избегание ненужных объектов:** Стараться создавать объекты только тогда, когда они действительно необходимы.

5.**Профилирование:** Использование инструментов профилирования для выявления узких мест (например, VisualVM).

6.**Управление размером кучи:** Настройка размера кучи (-Xms, -Xmx).

7.**Настройка сборщика мусора:** Выбор алгоритма сборки мусора в зависимости от нужд (например, G1, CMS).

8.**Использование пулов объектов:** Для управления большим количеством одинаковых объектов (например, Thread pool).

35. Инициализация переменных в JAVA. Способы инициализации переменных: по умолчанию, в конструкторах, через блоки инициализации. Примеры применения.

•**Инициализация переменных:** Присваивание значения переменной при ее объявлении или позже.

•**Способы:**

•**По умолчанию:**

•Примитивные типы: 0 (числовые типы), `false` (`boolean`), `\u0000` (`char`).

•Ссылочные типы: `null`.

•Происходит до вызова конструктора

```
class Example {
    int x;  // x инициализируется 0
    String str; //str инициализируется null
    }
```

•**При объявлении:**

```
    int y = 10;
    String message = "Hello";
```

•**В конструкторах:** Инициализация полей объекта в конструкторе.

```
class Product {
```

```
    String name;
    double price;
    public Product(String name, double price){
        this.name = name;
        this.price = price;
    }
}
```

•**Через блоки инициализации:** Как статические, так и нестатические блоки инициализации для общей инициализации, сокращения повторяющегося кода.

```
class MyClass{
    int id;
    String data;
    {
        this.id = generateId();
        this.data = "default";
    }
    static String version;
    static {
        version = "1.0";
    }
}
```

•**Примеры:**

```
class Example {
    int a;      //Инициализируется 0
    String str = "Hello"; //Инициализация при объявлении
    int b;
    {
        b = 10; //Инициализация в нестатическом блоке инициализации
    }
    static int staticNumber = 10; // Инициализация при объявлении
    static String staticMessage;
    static {
        staticMessage = "Static message"; //Инициализация в статическом блоке
    }
    public Example(int a){
        this.a = a; // Инициализация в конструкторе
    }
}
```

36. Математические функции. Класс `Math` в Java и его методы для выполнения вычислений. Примеры использования тригонометрических и экспоненциальных функций в задачах. Нужно ли создавать объект класса `Math` для использования математических методов.

•**Класс `Math`:**

•Предоставляет `static` методы для выполнения различных математических операций.

•Не нужно создавать объект класса для использования его методов.

•Находится в пакете `java.lang` (импортировать не нужно).

•**Методы:**

•**Основные:** `abs()`, `min()`, `max()`, `round()`, `sqrt()`, `pow()`, `random()`.

•**Тригонометрические:** `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`.

•**Экспоненциальные и логарифмические:** `exp()`, `log()`, `log10()`.

•**Константы:** `PI`, `E`.

•**Примеры:**

```
public static void main(String[] args) {
        double num = -10.5;
        double absValue = Math.abs(num); // Вывод: 10.5
        System.out.println("Absolute value of " + num + " is " + absValue);

        int minNum = Math.min(15, 8); // Вывод: 8
    System.out.println("Minimum of 15 and 8 is " + minNum);

    double power = Math.pow(2, 3); // Вывод: 8.0
        System.out.println("2 raised to 3 is " + power);

        double sqrt = Math.sqrt(16); // Вывод: 4.0
    System.out.println("Square root of 16 is " + sqrt);

        double angle = 45;
        double radians = Math.toRadians(angle);
        double sine = Math.sin(radians);
     System.out.println("Sine of 45 degrees is " + sine);

        double exp = Math.exp(2);
        System.out.println("e raised to 2 is " + exp);
    }
```

•**Нужно ли создавать объект:**

•**Нет:** `Math` - класс-утилита, все его методы `static`.

37. Абстракция и инкапсуляция класса. Понятие абстракции как отделения реализации класса от его использования. Как эти принципы улучшают структурирование кода и его модульность?

•**Абстракция:**

•Представление только существенной информации, скрытие деталей реализации.

•Сосредоточение на том, что делает объект, а не на том, как он это делает.

•Создание упрощенных моделей, которые могут быть использованы для решения определенных задач.

•**Инкапсуляция:** (как мы уже говорили ранее)

•Сокрытие внутреннего состояния объекта (данных) от внешнего мира.

•Предоставление доступа к данным через методы (геттеры и сеттеры).

•Защита данных от несанкционированного доступа и изменения.

•**Разница:**

•**Абстракция:** Фокусируется на что делает объект.

•**Инкапсуляция:** Фокусируется на как это делается и защищает данные.

•**Как улучшают:**

•**Структурирование:** Создание более организованного и понятного кода.

•**Модульность:** Разбиение системы на независимые модули, которые можно повторно использовать.

•**Изменение кода:** Легче вносить изменения в реализацию, не затрагивая код, который использует этот класс.

•**Упрощение:** Позволяет программистам использовать сложные системы, не вдаваясь в детали их реализации.

•**Безопасность:** Защита данных и логики.

•**Повышают гибкость и устойчивость:** Обеспечивают независимость и взаимодействие модулей друг с другом.

38. Отношения между классами. Основные виды отношений между классами: ассоциация, агрегация, композиция, наследование.

•**Отношения между классами:** Способы, которыми классы взаимодействуют друг с другом.

•**Виды:**

•**Ассоциация:** Простое отношение между классами, один класс использует другой.

•**Агрегация:** Отношение “has-a”, слабая связь, класс может существовать без класса, который им управляет.

•**Композиция:** Отношение “has-a”, сильная связь, класс не может существовать без класса, который им управляет.

•**Наследование:** Отношение “is-a”, создание подклассов на основе существующих классов.

39. Ассоциация. Понятие ассоциации как бинарного отношения между классами. Примеры реализации ассоциации в Java. Как ассоциация помогает моделировать взаимодействие объектов?

•**Ассоциация:**

•Связь между двумя классами, когда один класс использует другой класс.

•“Знает о другом” (один класс имеет ссылку на другой).

•Может быть однонаправленной (один класс знает о другом), или двунаправленной (оба класса знают друг о друге).

•Связь не является иерархической, и классы могут существовать независимо друг от друга.

•**Примеры реализации:**

```
class Student {
    private String name;
    private Course course; // Ассоциация: Student имеет Course
```

```
    public Student(String name) {
        this.name = name;
    }

    public void enrollCourse(Course course) {
        this.course = course;
    }

     public Course getCourse(){
        return course;
    }
}

class Course {
    private String name;
    public Course (String name){
        this.name = name;
    }

    public String getName(){
        return name;
    }
}

    public static void main(String[] args) {
          Course javaCourse = new Course("Java programming");
          Student student1 = new Student("John");
          student1.enrollCourse(javaCourse);

          System.out.println("Student course name: " +
student1.getCourse().getName()); // Student knows Course
    }
```

•Как помогает моделировать:

•Моделирование взаимодействия между объектами.

•Связь между классами в системе.

•Один класс использует функциональность или данные другого класса.

•Обеспечение гибкости взаимодействия.

40. Агрегация и композиция. Понятия агрегации и композиции, их различия. Как они отражают отношения «has-a» между объектами? Примеры реализации агрегации и композиции в проектировании классов.

•Агрегация:

•Отношение “has-a”, слабая связь.

•Один класс является контейнером для других классов.

•Контейнер может существовать без содержимого, и содержимое может существовать без контейнера.

•Пример: `Department` имеет `Employee`, но `Employee` может работать в другом `Department`.

•Композиция:

•Отношение “has-a”, сильная связь.

•Один класс является эксклюзивным контейнером для других классов.

•Контейнер и содержимое не могут существовать по отдельности.

•Пример: `Car` имеет `Engine`, `Engine` не может существовать без `Car`

•**Различия:**

•**Связь:** Агрегация - слабая, Композиция - сильная.

•**Жизненный цикл:** В агрегации объекты могут существовать независимо друг от друга; в композиции жизненный цикл связан (если родительский объект уничтожен, то дочерние тоже).

•**Отношение “has-a”:**

•Один класс имеет в качестве полей объекты других классов.

•**Примеры реализации:**

```
// Агрегация
class Department {
    private List<Employee> employees;

    public Department(List<Employee> employees){
        this.employees = employees;
    }
    public void addEmployee(Employee employee){
         employees.add(employee);
    }
}

class Employee {
    String name;

    public Employee (String name){
         this.name = name;
    }
}

// Композиция
class Car {
    private Engine engine;

    public Car() {
         this.engine = new Engine(); // Engine создается внутри Car
    }

    class Engine {
        }

    }


 public static void main(String[] args) {
         List<Employee> employeesList = new ArrayList<>();
        employeesList.add(new Employee("John Doe"));
        Department department = new Department(employeesList); //Агрегация

         Car car = new Car(); //Композиция
```

```
}
```

41. Обработка примитивных типов как объектных. Использование классов-оберток для работы с примитивными типами как с объектами. Примеры преобразования примитивных типов в объекты и обратно.

- **Классы-обертки:** (уже обсуждали ранее, но повторим для контекста)
- Предоставляют объектное представление примитивных типов.
- `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`.
- Позволяют использовать примитивы как объекты (например, в коллекциях).
- **Преобразование примитивов в объекты (упаковка, boxing):**
- Используем конструктор класса-обертки.
- Или используем `valueOf()` метод.

```
int num = 10;
Integer numObj = new Integer(num); // Способ 1 (устаревший)
Integer numObj2 = Integer.valueOf(num); // Способ 2 (рекомендуемый)

double price = 19.99;
Double priceObj = new Double(price); // Способ 1 (устаревший)
Double priceObj2 = Double.valueOf(price); // Способ 2 (рекомендуемый)

boolean status = true;
Boolean statusObj = new Boolean(status);
Boolean statusObj2 = Boolean.valueOf(status);

char ch = 'A';
Character chObj = new Character(ch);
Character chObj2 = Character.valueOf(ch);
```

- **Преобразование объектов в примитивы (распаковка, unboxing):**
- Используем методы `intValue()`, `doubleValue()`, `booleanValue()`, `charValue()` и т.д.

```
Integer numObj = Integer.valueOf(10);
int num = numObj.intValue();

Double priceObj = Double.valueOf(19.99);
double price = priceObj.doubleValue();

Boolean statusObj = Boolean.valueOf(true);
boolean status = statusObj.booleanValue();

Character chObj = Character.valueOf('A');
char ch = chObj.charValue();
```

- **Примеры:**

```
public static void main(String[] args) {
        // Примитив -> Объект
        int intValue = 42;
    Integer intObject = Integer.valueOf(intValue);
```

```
        System.out.println("Integer object: " + intObject);

        char charValue = 'B';
        Character charObject = Character.valueOf(charValue);
        System.out.println("Character object: " + charObject);

        // Объект -> Примитив
        int intValue2 = intObject.intValue();
        System.out.println("Primitive int value: " + intValue2);

        char charValue2 = charObject.charValue();
        System.out.println("Primitive char value: " + charValue2);
    }
```

42. Классы-обертки. Основные возможности классов-оберток: `Integer`, `Double`, `Boolean` и других. Методы для преобразования значений и сравнения объектов. Примеры использования методов `parseInt`, `valueOf` и `compareTo`.

- **Классы-обертки (основные):**
- `Integer`: `int`
- `Double`: `double`
- `Boolean`: `boolean`
- `Long`: `long`
- `Float`: `float`
- `Character`: `char`
- `Byte`: `byte`
- `Short`: `short`
- **Основные возможности:**
- **Создание объекта:** конструктор или статические методы `valueOf()`.
- **Получение примитивного значения:** `intValue()`, `doubleValue()`, `booleanValue()`, `longValue()`, `floatValue()`, `charValue()`, `byteValue()`, `shortValue()`.
- **Преобразование строк в числа:** `parseInt()`, `parseDouble()`, `parseBoolean()`, и т.д.
- **Сравнение объектов:** `equals()`, `compareTo()`.
- **Другие методы:** `toString()`, `hashCode()`, `toBinaryString()`, `toHexString()`, и т.д.
- **Примеры:**

```
    public static void main(String[] args) {
        // Создание объектов
        Integer numObj1 = Integer.valueOf(10);
        Integer numObj2 = Integer.valueOf("20");
        Double priceObj = Double.valueOf(3.14);
        Boolean statusObj = Boolean.valueOf(true);

        // Преобразование строк в числа
        int numFromString = Integer.parseInt("123");
```

```
        System.out.println("numFromString: " + numFromString);
         double doubleFromString = Double.parseDouble("45.6");
        System.out.println("doubleFromString: " + doubleFromString);

        // Сравнение объектов
        Integer numObj3 = Integer.valueOf(10);
        System.out.println("numObj1 equals numObj3? " +
numObj1.equals(numObj3)); // Вывод: true
        System.out.println("numObj1 compared to numObj2? " +
numObj1.compareTo(numObj2)); // Вывод: -1
      System.out.println("numObj2 compared to numObj1? " +
numObj2.compareTo(numObj1)); // Вывод: 1

        // Получение примитивных значений
        int numValue = numObj1.intValue();
      double priceValue = priceObj.doubleValue();
      boolean statusValue = statusObj.booleanValue();

      System.out.println("numValue: " + numValue + ", priceValue: " + priceValue
+ ", statusValue: " + statusValue);
  }
```

43. Автоматическое преобразование. Что такое автоупаковка (autoboxing) и автораспаковка (unboxing) в Java? Как они автоматически преобразуют значения примитивных типов в объекты и обратно? Примеры использования.

- **Автоупаковка (Autoboxing):**
- Автоматическое преобразование примитивного типа в соответствующий ему объект-обертку.
- Происходит, когда примитивное значение присваивается переменной ссылочного типа (обертке).
- **Автораспаковка (Unboxing):**
- Автоматическое преобразование объекта-обертки в соответствующий ему примитивный тип.
- Происходит, когда объект-обертка присваивается переменной примитивного типа или используется в контексте, где требуется примитив.
- **Как работает:**
- Компилятор автоматически вставляет вызовы методов `valueOf()` для автоупаковки и методы `intValue()`, `doubleValue()`, и т.д. для автораспаковки.
- **Примеры:**

```
 public static void main(String[] args) {
        // Автоупаковка (autoboxing)
       int intValue = 10;
        Integer intObject = intValue; // int -> Integer
        System.out.println("Integer object: " + intObject);

      double doubleValue = 3.14;
        Double doubleObject = doubleValue; // double -> Double
```

```
        System.out.println("Double object: " + doubleObject);

        boolean status = true;
        Boolean statusObject = status; // boolean -> Boolean
        System.out.println("Boolean object: " + statusObject);

        // Автораспаковка (unboxing)
        int intValue2 = intObject; // Integer -> int
        System.out.println("int value 2: " + intValue2);

        double doubleValue2 = doubleObject; // Double -> double
        System.out.println("double value 2: " + doubleValue2);

        boolean status2 = statusObject; // Boolean -> boolean
        System.out.println("status value 2: " + status2);


        // Autoboxing при использовании в коллекциях
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1); //Автоупаковка int -> Integer
        int num = numbers.get(0); //Автораспаковка Integer -> int
        System.out.println("Number from list " + num);
    }
```

- **Когда использовать:**
- Упрощает код, нет необходимости явно преобразовывать.
- При работе с коллекциями, которые хранят только объекты.
- Операции с обертками и примитивами в одном выражении.
- **Предостережение:** Может вызывать `NullPointerException`, если обертка `null` и используется в контексте автораспаковки.

44. Класс `String`. Понятие неизменяемости(иммутабельности) строк в Java. Как создаются объекты типа `String`? Примеры работы с методами создания, сравнения и модификации строк.

- **Класс `String`:**
- Представляет последовательность символов (текстовую строку).
- `String` — **неизменяемый (immutable) класс:** После создания объект `String` нельзя изменить (нельзя изменить его состояние).
- Когда вы меняете строку, создается новый объект `String` в памяти.
- **Создание объектов `String`:**
- **Строковый литерал:**

```
String str1 = "Hello"; // Объект создается в String pool
String str2 = "Hello"; // str2 ссылается на тот же объект в String pool, что и str1
```

- **Конструктор:**

```
String str3 = new String("World"); // Создается новый объект в куче
```

```
 String str4 = new String("World"); // Создается новый объект в куче, отличный от str3
```

- **Методы:**
- **Создание:** `new String()`, строковые литералы, `valueOf()` (для преобразования других типов в строки).
- **Сравнение:** `equals()`, `equalsIgnoreCase()`, `compareTo()`.
- **Модификация:** (неизменяемость - значит, что создается новая строка) `concat()`, `substring()`, `trim()`, `replace()`, `toUpperCase()`, `toLowerCase()`, и т.д.
- **Поиск:** `indexOf()`, `lastIndexOf()`, `startsWith()`, `endsWith()`, `contains()`.
- **Доступ к символам:** `charAt()`, `length()`.
- **Примеры:**

```
 public static void main(String[] args) {
        String str1 = "Hello";
       String str2 = "world";
        System.out.println("String 1 " + str1 + ", String 2 " + str2);
         //Создание строк
       String str3 = new String("Hello");
     System.out.println("String 3 " + str3);

       // Сравнение строк
        System.out.println("str1 equals str3? " + str1.equals(str3)); // true
        System.out.println("str1 == str3? " + (str1 == str3)); // false

        String str4 = "hello";
        System.out.println("str1 equals str4(ignore case)? " + str1.equalsIgnoreCase(str4)); //true

      System.out.println("str1 compare to str2? " + str1.compareTo(str2)); // отрицательное число


       // Модификация строк (создается новый объект)
     String str5 = str1.concat(" ").concat(str2);
    System.out.println("Concat string: " + str5); // Hello world
     String substring = str5.substring(0, 5);
    System.out.println("SubString: " + substring); // Hello
       String upperCaseStr = str1.toUpperCase();
    System.out.println("Upper case string: " + upperCaseStr); // HELLO

       // Поиск в строках
     int index = str5.indexOf("w");
      System.out.println("Index of 'w' " + index); // 6
     System.out.println("String starts with 'H'? " + str5.startsWith("H")); // true

        System.out.println("Character on index 1 " + str1.charAt(1)); //e
    }
```

45. Строки в JAVA. Замена и разделение строк. Методы класса `String` для замены символов и разделения строк. Примеры работы с методами `replace` и `split`.

•**Методы класса `String`:**

•`replace(char oldChar, char newChar)`: Заменяет все вхождения символа `oldChar` на `newChar`.

•`replace(CharSequence target, CharSequence replacement)`: Заменяет все вхождения подстроки `target` на `replacement`.

•`replaceAll(String regex, String replacement)`: Заменяет все вхождения подстроки, соответствующие регулярному выражению, на `replacement`.

•`split(String regex)`: Разделяет строку на массив подстрок, используя разделитель `regex`.

•`split(String regex, int limit)`: Разделяет строку на массив подстрок, используя разделитель `regex` и ограничивая количество подстрок.

•**Примеры:**

```
    public static void main(String[] args) {
        String str = "Hello, World! Hello, Java!";

        // Замена символов
        String replacedChar = str.replace('l', 'L');
        System.out.println("Replaced char: " + replacedChar); // "HeLLo, WorLd! HeLLo, Java!"

        // Замена подстроки
        String replacedSubstr = str.replace("Hello", "Hi");
        System.out.println("Replaced substr: " + replacedSubstr); // "Hi, World! Hi, Java!"

        // Замена подстроки с использованием регулярного выражения
        String replacedAll = str.replaceAll("Hello", "Hola");
        System.out.println("Replaced All " + replacedAll);

        String replacedWithRegex = str.replaceAll("[HW]", "Z"); //Hello -> Zello
        System.out.println("Replaced with regex: " + replacedWithRegex); //"Zello, Zorl! Zello, Java!"

        // Разделение строки на массив
        String str2 = "apple,banana,cherry";
        String[] parts = str2.split(",");
        System.out.print("Split: ");
        for (String part : parts) {
            System.out.print(part + " ");  // apple banana cherry
        }
        System.out.println();

        // Разделение строки с ограничением
        String str3 = "one:two:three:four";
        String[] limitParts = str3.split(":", 2);
        System.out.print("Split with limit: ");
        for (String part : limitParts) {
            System.out.print(part + " ");  // one two:three:four
        }
        System.out.println();
    }
```

46. Строки в JAVA. Преобразования между строками и массивами. Как преобразовать строку в массив символов и наоборот? Примеры использования методов `toCharArray` и `valueOf`.

•Преобразование строки в массив символов:

•`toCharArray()`: Метод класса `String`, преобразует строку в массив `char[]`.

•Преобразование массива символов в строку:

•`String.valueOf(char[] data)`: Статический метод класса `String`, создает строку из массива `char[]`.

•`new String(char[] data)`: Создает строку из массива `char[]`.

•Примеры:

```
    public static void main(String[] args) {
        String str = "Hello";

        // Строка -> Массив символов
        char[] charArray = str.toCharArray();

        System.out.print("String to char array: ");
        for (char ch : charArray) {
            System.out.print(ch + " "); // H e l l o
        }
        System.out.println();

        // Массив символов -> Строка
        char[] charArray2 = {'W', 'o', 'r', 'l', 'd'};
        String str2 = String.valueOf(charArray2);
        System.out.println("Char array to string: " + str2); // "World"
        String str3 = new String(charArray2);
        System.out.println("Char array to string: " + str3); // "World"
    }
```

47. Строки в JAVA. Класс `StringBuilder` и `StringBuffer`. Понятие изменяемых строк. Основные отличия между `StringBuilder` и `StringBuffer`. Примеры их использования. Влияние классов `StringBuilder` и `StringBuffer` на типобезопасность.

•Изменяемые строки:

•Объекты, состояние которых можно изменять после создания.

•`StringBuilder` и `StringBuffer` - классы для работы с изменяемыми строками.

•**`StringBuilder`:**

•Непотокобезопасный класс для работы с изменяемыми строками.

•Используется в однопоточных программах.

•Более производительный, чем `StringBuffer`.

•**`StringBuffer`:**

•Потокобезопасный класс для работы с изменяемыми строками.

•Используется в многопоточных программах.

•Менее производительный, чем `StringBuilder`.

•**Основные отличия:**

•**Потокобезопасность:** `StringBuilder` - не потокобезопасен, `StringBuffer` - потокобезопасен (синхронизированные методы).

•**Производительность:** `StringBuilder` быстрее, чем `StringBuffer`.

•**Область использования:** `StringBuilder` - в однопоточных, `StringBuffer` - в многопоточных программах.

•**Примеры:**

```
    public static void main(String[] args) {
        // StringBuilder
        StringBuilder strBuilder = new StringBuilder();
        strBuilder.append("Hello").append(" ").append("World");
        System.out.println("String builder: " + strBuilder);
        strBuilder.insert(5, ","); //Hello, World
        System.out.println("String builder after insert: " + strBuilder);
        strBuilder.delete(0,2);
        System.out.println("String builder after delete: " + strBuilder);

        // StringBuffer
        StringBuffer strBuffer = new StringBuffer();
        strBuffer.append("Java").append(" ").append("Code");
        System.out.println("String buffer " + strBuffer);
        strBuffer.reverse();
        System.out.println("Reversed buffer " + strBuffer);
    }
```

•**Влияние на типобезопасность:**

•`StringBuilder` и `StringBuffer` не влияют на типобезопасность.

48. Строки в JAVA. Преобразование символов и чисел в строки. Какие методы используются для преобразования чисел, символов и объектов в строки? Примеры работы с методами `String.valueOf()` и `toString()`.

•**Методы преобразования в строку:**

•`String.valueOf(type value)`: Статический метод класса `String`, преобразует разные типы (примитивы, объекты) в строки.

•`object.toString()`: Метод класса `Object`, унаследованный всеми классами, преобразует объект в строковое представление.

•**Примеры:**

```
class Person {
  String name;
  int age;
    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
     @Override
   public String toString(){
```

```
        return "Person{name=" + name + ", age=" + age +"}";
    }
}
 public static void main(String[] args) {
        int num = 123;
        double price = 99.99;
        char ch = 'X';
    boolean status = true;

        // Преобразование примитивов в строку
        String numStr = String.valueOf(num);
        String priceStr = String.valueOf(price);
        String charStr = String.valueOf(ch);
       String statusStr = String.valueOf(status);
        System.out.println("Number: " + numStr + ", Price " + priceStr + ", Char:
" + charStr + " ,Status: " + statusStr);

        // Преобразование объекта в строку
        Person person = new Person("John", 30);
        String personStr = person.toString(); // вызывается переопределенный
метод toString
        System.out.println("Person: " + personStr);
        String personStr2 = String.valueOf(person);  // valueOf вызывает метод
toString
        System.out.println("Person2: " + personStr2);
 }
```

- **`String.valueOf()`:**
- Работает со всеми примитивами и объектами.
- Для объектов вызывает метод `toString()`.
- **`toString()`:**
- Наследован от `Object`.
- Рекомендуется переопределять в классах, чтобы получить информативное представление объекта.

49. Строки в JAVA. Интернированные строки. Что такое интернированные строки? Как JVM оптимизирует работу с повторяющимися строками? Примеры их использования.

- **Интернированные строки:**
- Строковые объекты, которые хранятся в специальном пуле (String pool) в памяти.
- JVM создает только один экземпляр каждой уникальной строковой константы.
- Все ссылки на одинаковые строковые константы указывают на один и тот же объект.
- **Оптимизация:**
- JVM проверяет, есть ли уже такая строка в пуле.
- Если есть, то возвращается ссылка на существующий объект.
- Если нет, создается новый объект в пуле и возвращается ссылка.
- Экономия памяти (уменьшает количество дубликатов).

•Ускоряет сравнение строк (сравниваются ссылки, а не содержимое).

•Как работают:

•Строковые литералы (“Hello”) автоматически интернируются.

•Метод `String.intern()` позволяет вручную добавить строку в пул.

•Примеры:

```
public static void main(String[] args) {
    String str1 = "Hello"; // интернированная строка (пул)
    String str2 = "Hello"; // интернированная строка (ссылка на str1)
    String str3 = new String("Hello"); // объект в куче
    String str4 = new String("Hello").intern(); // интернированная строка (добавление в пул)

    System.out.println("str1 == str2? " + (str1 == str2)); // true (ссылка на один объект)
    System.out.println("str1 == str3? " + (str1 == str3)); // false (разные объекты)
    System.out.println("str1 == str4? " + (str1 == str4)); // true (ссылка на один объект)

    String str5 = new String("New String");
    String str6 = new String("New String");
    System.out.println("str5 equals str6? " + str5.equals(str6)); // true (сравнение значений)
    System.out.println("str5 == str6? " + (str5 == str6)); // false (сравнение ссылок)
    System.out.println("str5 == str6.intern()? " + (str5 == str6.intern())); // false (интернирование 6)
    String str7 = str5.intern();
    String str8 = str6.intern();
    System.out.println("str7 == str8? " + (str7 == str8)); // true (интернированные строки ссылаются на один объект)

}
```

50. Наследование в JAVA. Основные принципы наследования в Java. Что такое суперклассы(родительские) и подклассы(дочерние)? Как наследование помогает переиспользовать код? Примеры реализации наследования.

•Наследование:

•Механизм, позволяющий одному классу (подклассу) наследовать свойства (поля) и поведение (методы) другого класса (суперкласса).

•Отношение “is-a”.

•Создание иерархии классов.

•Суперкласс (родительский класс):

•Класс, свойства и поведение которого наследуются другими классами.

•Подкласс (дочерний класс):

•Класс, который наследует свойства и поведение суперкласса.

•Может добавлять новые поля и методы, а также переопределять методы суперкласса.

•**Принципы:**

•**Переиспользование кода:** Позволяет не дублировать код, а использовать его в подклассах.

•**Расширяемость:** Подклассы могут расширять функциональность суперкласса.

•**Полиморфизм:** Объекты подклассов могут быть использованы в контексте суперкласса (динамическое связывание).

•**Реализация:**

•Используется ключевое слово `extends`.

```
class Animal {
    String name;
    public Animal(String name){
        this.name = name;
    }
    public void makeSound(){
        System.out.println("Generic animal sound");
    }
}

class Dog extends Animal { // Dog наследует Animal
    String breed;
    public Dog(String name, String breed){
        super(name); // вызов конструктора суперкласса
        this.breed = breed;
    }

    @Override
    public void makeSound(){
        System.out.println("Woof!"); // Переопределенный метод
    }
}
class Cat extends Animal{
    public Cat(String name){
        super(name);
    }
    @Override
    public void makeSound(){
        System.out.println("Meow!");
    }
}

public static void main(String[] args) {
    Dog dog1 = new Dog("Buddy", "Labrador");
    System.out.println("Dog name: " + dog1.name + ", breed: " + dog1.breed);
    dog1.makeSound();

    Cat cat1 = new Cat("Whiskers");
    cat1.makeSound();
}
```

•**Примеры переиспользования кода:**

•Общие поля и методы (например, `Animal` и его подклассы).
•Реализация интерфейсов.

51. Перегрузка метода в Java (overload). Переопределение метода в Java (override). В чем разница между перегрузкой и переопределением методов.

•**Перегрузка (Overload):**
•Создание нескольких методов с одним и тем же именем в одном классе.
•Методы должны иметь разные списки параметров (разное количество или тип параметров).
•Компилятор выбирает, какой метод вызывать, на основе типов аргументов.
•Перегруженные методы могут иметь разные возвращаемые значения.
•Перегрузка происходит в одном классе.
•**Переопределение (Override):**
•Переопределение метода, унаследованного от суперкласса в подклассе.
•Метод подкласса должен иметь то же имя, тот же список параметров и тот же возвращаемый тип, что и метод суперкласса.
•Переопределенный метод должен иметь модификатор доступа не менее доступный чем метод суперкласса.
•Переопределение происходит между суперклассом и подклассом.
•Аннотация `@Override` рекомендуется для явного указания переопределенного метода.
•**Разница:**
•**Где:** Перегрузка в одном классе, переопределение между суперклассом и подклассом.
•**Параметры:** Перегрузка — разные параметры, переопределение — одинаковые параметры.
•**Возвращаемый тип:** Перегрузка - может быть разным, переопределение - должен совпадать.
•**Назначение:** Перегрузка - создание методов с одинаковым названием для разного поведения, переопределение - изменение поведения метода в подклассе.

```
class MathUtils {
    // перегруженные методы
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b){
        return a + b;
    }
}
class Shape {
    public void draw(){
        System.out.println("Drawing a shape");
```

```
    }
}
class Circle extends Shape{
    @Override
    public void draw(){
        System.out.println("Drawing a circle"); // Переопределение метода
    }
}
public static void main(String[] args) {
        MathUtils math = new MathUtils();
        System.out.println(math.add(2,3)); //Вызов перегруженного метода add(int a, int b)
        System.out.println(math.add(2.5,3.5)); //Вызов перегруженного метода add(double a, double b)
        Shape shape1 = new Shape();
        shape1.draw(); // Drawing a shape
        Shape shape2 = new Circle();
        shape2.draw(); // Drawing a circle (динамическое связывание)
    }
```

52. Наследование и отношение `is-a`. Как наследование реализует отношение «is-a»? Когда использование наследования может быть нецелесообразным? Примеры решений.

- **Отношение “is-a”:**
- Означает, что объект одного класса является частным случаем другого класса.
- Реализуется через наследование.
- Например, `Dog` is-a `Animal`, `Circle` is-a `Shape`.
- **Как реализуется:**
- Подкласс наследует свойства и методы суперкласса.
- Подкласс может использоваться в любом месте, где ожидается суперкласс (полиморфизм).
- `Dog` унаследовал `name` и `makeSound()` метод от `Animal`, но также имеет свой `breed`
- **Нецелесообразное использование:**
- **Неправильная иерархия:** Если между классами нет логической связи “is-a”, то наследование не подходит.
- **Проблемы с сопровождением:** При большом количестве уровней наследования код становится сложным для понимания.
- **Нарушение принципа единственной ответственности:** Когда класс наследует много разных свойств, код становится менее гибким и перегруженным.
- **Хрупкость:** Любое изменение в суперклассе может повлиять на подклассы.
- **Примеры решений:**
- **Использование композиции вместо наследования:**
- Когда не подходит отношение “is-a”, используйте отношение “has-a”.
- Например, `Car` has-a `Engine`.
- **Использование интерфейсов:**

•Определить контракт для классов, которые должны иметь определенный функционал.
•Классы могут реализовывать несколько интерфейсов.

```
class Engine {
    void start() { /* ... */ }
}
class Car {
    Engine engine;
    public Car(){
        this.engine = new Engine();
    }
    void start(){
     this.engine.start();
    }
}
   interface Flyable {
      void fly();
    }
    class Bird implements Flyable {
     @Override
     public void fly(){
            System.out.println("Bird is flying");
     }
    }
   class Plane implements Flyable {
    @Override
     public void fly(){
           System.out.println("Plane is flying");
     }
   }

    public static void main(String[] args) {
       // Плохо. Наследование не подходит т.к. Square не is-a Shape (не может
переопределить)
       // class Square extends Shape {
       //       void draw(){
       //           System.out.println("Drawing a square"); // неверная логика
       //     }
       //}
       //Решение - композиция
         Car car = new Car();
        car.start();
        //Решение - интерфейсы
       Flyable bird = new Bird();
       bird.fly();
       Flyable plane = new Plane();
       plane.fly();
    }
```

53. Ключевое слово `super`. Роль ключевого слова `super` в Java. Использование для вызова методов и конструкторов суперкласса. Примеры реализации.

•Ключевое слово `super`:

•Ссылка на объект суперкласса (родительского класса).
•Используется в подклассе для доступа к членам суперкласса, которые переопределены в подклассе.
•Используется для вызова конструктора суперкласса.
•**Использование:**

•**Вызов конструктора суперкласса:** `super()` (должен быть первым оператором в конструкторе подкласса).
•**Доступ к методам суперкласса:** `super.methodName()`.
•**Доступ к полям суперкласса:** `super.fieldName`.
•**Примеры:**

```
class Animal {
  String name;
  public Animal(String name){
    this.name = name;
  }
  public void makeSound(){
    System.out.println("Generic animal sound");
  }
}

class Dog extends Animal {
  String breed;
  public Dog(String name, String breed) {
    super(name);  // Вызов конструктора Animal
    this.breed = breed;
  }
  @Override
  public void makeSound(){
    super.makeSound();
    System.out.println("Woof!"); // Вызов метода суперкласса и переопределение
  }

    public void printName() {
          System.out.println("Dog name from parent: " + super.name); // доступ к полю суперкласса
    }
}


public static void main(String[] args) {
  Dog dog1 = new Dog("Buddy", "Labrador");
  dog1.makeSound();  // Выведет: "Generic animal sound", "Woof!"
  dog1.printName(); // Выведет: "Dog name from parent: Buddy"
}
```

54. Цепочка конструкторов. Понятие цепочки конструкторов. Как вызвать один конструктор из другого с использованием `this()` и `super()`? Примеры реализации.

•**Цепочка конструкторов:**

•Механизм, когда один конструктор вызывает другой конструктор в том же классе или в суперклассе.
•Позволяет избежать дублирования кода инициализации.
•**`this()`:**

•Вызывает другой конструктор этого же класса.
•Должен быть первым оператором в конструкторе.
•**`super()`:**

•Вызывает конструктор суперкласса.
•Должен быть первым оператором в конструкторе.
•**Правила:**

•В конструкторе можно использовать либо `this()`, либо `super()`, но не оба одновременно.
•Если `this()` или `super()` не указаны, то вызывается конструктор суперкласса без параметров (по умолчанию).
•**Примеры:**

```
class Animal {
  String name;
   int age;
  public Animal() {
       System.out.println("Animal default constructor");
  }
  public Animal(String name) {
     this(); // Вызов конструктора Animal()
       System.out.println("Animal name constructor " + name);
       this.name = name;
  }
public Animal(String name, int age){
       this(name); // Вызов конструктора Animal(String name)
     this.age = age;
       System.out.println("Animal name age constructor " + name + ", age: " +
age);
  }
}

class Dog extends Animal {
  String breed;
  public Dog(String name, String breed) {
     super(name, 3);  // Вызов конструктора Animal(String name, int age)
        System.out.println("Dog constructor " + name + ", breed " + breed);
     this.breed = breed;

  }
  public Dog(){
     super(); //вызов Animal()
  }
}


public static void main(String[] args) {
```

```
    Dog dog1 = new Dog("Buddy", "Labrador"); // Вызовет: Animal default constructor, Animal name constructor, Animal age constructor, Dog constructor
    System.out.println();
    Dog dog2 = new Dog(); // Вызов Animal()
}
```

55. Класс `Object` и его основные методы. Роль класса `Object` как суперкласса для всех классов в Java. Как метод `toString()` используется для представления объекта в виде строки? Примеры переопределения метода.

•Класс `Object`:

•Корневой класс иерархии классов в Java.
•Все классы неявно наследуются от `Object` (даже если явно не указано `extends Object`).
•Обеспечивает общие методы для всех объектов.
•Основные методы:

•`toString()`: Возвращает строковое представление объекта.
•`equals(Object obj)`: Сравнивает объекты на равенство.
•`hashCode()`: Возвращает хеш-код объекта.
•`getClass()`: Возвращает класс объекта.
•`clone()`: Создает копию объекта.
•`notify()`, `notifyAll()`, `wait()`: Методы для работы с многопоточностью.
•`finalize()`: Метод вызывается перед удалением объекта сборщиком мусора (устарел, deprecated).
•`toString()`:

•По умолчанию возвращает имя класса и хеш-код объекта (например, `MyClass@123abc`).
•Рекомендуется переопределять в пользовательских классах, чтобы возвращать более информативное строковое представление объекта.
•Примеры переопределения `toString()`:

```
class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Переопределение метода toString()
    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
```

```
    @Override
    public boolean equals(Object obj){
        if(this == obj) return true;
        if(obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return age == person.age && Objects.equals(name, person.name);
    }

    @Override
    public int hashCode(){
        return Objects.hash(name, age);
  }
}

public static void main(String[] args) {
  Person person = new Person("John Doe", 30);
  System.out.println(person); // Вызовет переопределенный toString()
 Person person2 = new Person("John Doe", 30);
 System.out.println("Equals objects " + person.equals(person2));
System.out.println("HashCode person " + person.hashCode());
    System.out.println("HashCode person2 " + person2.hashCode());

}
```

56. Полиморфизм. Понятие полиморфизма в Java. Как переменная супертипа может ссылаться на объект подтипа? Примеры применения полиморфизма для создания гибкого кода.

•Полиморфизм (Polymorphism):

•“Множество форм” - способность объекта принимать различные формы.
•В Java реализуется через наследование и интерфейсы.
•Позволяет обрабатывать объекты разных классов как объекты общего суперкласса или интерфейса.

•Переменная супертипа ссылается на объект подтипа:

•Объект подкласса (дочернего класса) может быть присвоен переменной типа суперкласса (родительского класса).
•Это возможно благодаря наследованию (подкласс “is-a” суперкласс).
•При вызове метода у такой переменной будет вызываться метод из объекта подкласса (динамическое связывание).

•Примеры применения:

```
 class Animal {
    public void makeSound() {
        System.out.println("Generic animal sound");
    }
  }
```

```
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}

public static void main(String[] args) {
    Animal animal1 = new Dog(); // полиморфизм
    Animal animal2 = new Cat(); // полиморфизм
    animal1.makeSound(); // Выведет "Woof!" (динамическое связывание)
    animal2.makeSound(); // Выведет "Meow!" (динамическое связывание)

    //Создание массива Animal, где можно хранить объекты любых классов, которые являются подтипом Animal.
    Animal[] animals = new Animal[3];
    animals[0] = new Dog();
    animals[1] = new Cat();
    animals[2] = new Animal();
    for(Animal animal: animals){
        animal.makeSound();  // Вызов нужного метода для каждого конкретного объекта
    }

}
```

•Польза:

•Гибкость: Можно обрабатывать разные объекты одним и тем же кодом.

•Расширяемость: Легко добавлять новые классы, не меняя существующий код.

•Уменьшение дублирования кода: Упрощает разработку и поддержку.

•Полиморфный вызов методов: Во время выполнения определяется метод какого класса будет вызываться.

57. Интерфейсы в Java. Понятие интерфейсов как конструкций для определения общих операций. Основные элементы интерфейсов: константы и абстрактные методы. Примеры использования интерфейсов для создания обобщенных решений.

•Интерфейсы:

•Конструкции, которые определяют контракт, набор абстрактных методов, которые должны быть реализованы классами.
•Не содержат реализации (только описание поведения).
•Используются для достижения полиморфизма и создания обобщенных решений.

•**Основные элементы:**

•**Константы:** `public static final` (по умолчанию).

•**Абстрактные методы:** `public abstract` (по умолчанию, без реализации).

•**Примеры использования:**

```
interface Drawable {
    double PI = 3.14; // Константа
    void draw(); // Абстрактный метод
}

class Circle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Square implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a square");
    }
}

public static void main(String[] args) {
    Drawable drawable1 = new Circle(); // полиморфизм
    Drawable drawable2 = new Square(); // полиморфизм
    drawable1.draw(); // Выведет "Drawing a circle"
    drawable2.draw(); // Выведет "Drawing a square"

    //Массив интерфейсов
    Drawable[] drawables = new Drawable[2];
    drawables[0] = new Circle();
    drawables[1] = new Square();
    for(Drawable drawable: drawables){
        drawable.draw();
    }

}
```

•**Польза:**

•**Определение контракта:** Гарантия, что классы реализуют определенную функциональность.

•**Множественное наследование типа:** Класс может реализовывать несколько интерфейсов.

•**Полиморфизм:** Возможность использовать объекты разных классов как объекты одного интерфейсного типа.

•**Слабая связь:** Уменьшение зависимостей между классами.

•**Гибкость:** Классы, реализующие интерфейс, могут иметь свою собственную реализацию методов.

58. Интерфейсы в Java. Понятие интерфейсов как конструкций для определения общих операций. Особенности интерфейсов, добавленные в JAVA 8 версии. Дефолтные методы в интерфейсах.

•**Интерфейсы:** (повторение)

•Определяют контракт для классов.
•Содержат абстрактные методы (без реализации).
•**Особенности Java 8:**

•**Default методы:**
•Методы с реализацией в интерфейсе (используется ключевое слово `default`).
•Позволяют добавлять новую функциональность в интерфейсы без необходимости изменения классов, которые их реализуют.
•Классы могут переопределять дефолтные методы.

```
interface Logger {
    void log(String message);

    default void logError(String message){
     log("ERROR: " + message);
    }
}
class ConsoleLogger implements Logger {
  @Override
  public void log(String message) {
      System.out.println("Console: " + message);
    }

}
 class FileLogger implements Logger {
  @Override
  public void log(String message) {
      System.out.println("File: " + message);
   }
     // Переопределение default метода
  @Override
    public void logError(String message){
          log("FILE ERROR: " + message);
    }
}

public static void main(String[] args) {
    Logger logger1 = new ConsoleLogger();
    logger1.log("Hello");
    logger1.logError("Wrong login"); // вызов default метода

     Logger logger2 = new FileLogger();
    logger2.log("World");
    logger2.logError("Error in file"); // вызов переопределенного default метода
  }
```

•**Польза:**

•**Обратная совместимость:** Добавляйте новые методы в интерфейсы без нарушения старого кода.
•**Уменьшение дублирования кода:** Общая реализация для всех реализующих классов.
•**Эволюция API:** Позволяют развивать интерфейсы со временем.

59. Интерфейсы в Java. Особенности интерфейсов. Чем интерфейсы отличаются от классов? Как используются ключевые слова `interface` и `implements`? Примеры объявления и реализации интерфейсов.

•**Интерфейсы:** (повторение)

•Определяют контракт (набор методов).
•Описывают, что должны делать классы, но не как.
•Могут содержать константы, абстрактные методы (до Java 8), default методы (с Java 8) и статические методы(с Java 8).
•**Отличия от классов:**

•**Нельзя создавать объекты:** Интерфейсы нельзя инстанцировать (нельзя создать объект интерфейсного типа).
•**Нет реализации:** Интерфейсы содержат только объявления методов (кроме default и static).
•**Множественное наследование типа:** Класс может реализовывать несколько интерфейсов, но может наследовать только один класс.
•Классы описывают что и как, интерфейсы только что.
•**Ключевые слова:**

•`interface`: Используется для объявления интерфейса.
•`implements`: Используется в классе для указания, что он реализует интерфейс (должен реализовать все его абстрактные методы).
•**Примеры:**

```
interface Movable {
    int MAX_SPEED = 100; // Константа
    void move(); // Абстрактный метод
}

interface Swimmable {
    void swim();
}

class Car implements Movable {
    @Override
    public void move() {
        System.out.println("Car is moving");
    }
```

```
}

    class Boat implements Movable, Swimmable {
    @Override
    public void move() {
       System.out.println("Boat is moving");
    }
  @Override
    public void swim() {
      System.out.println("Boat is swimming");
      }
    }

  public static void main(String[] args) {
    Movable movable1 = new Car();
    movable1.move();
       System.out.println("Max speed is " + Movable.MAX_SPEED);
     Movable movable2 = new Boat();
       movable2.move();
    Swimmable swimmable = new Boat();
    swimmable.swim();
  }
```

60. Интерфейсы в Java 8 и 9. Новые возможности интерфейсов, такие как `default` и `static` методы (Java 8), а также `private` и `private static` методы (Java 9). Примеры реализации и применения.

•Java 8:

•`default` методы: (уже обсуждали)

•Реализация метода в интерфейсе.

•Позволяют добавлять методы в интерфейс без нарушения старого кода.

•Реализующие классы могут переопределять дефолтные методы.

•`static` методы:

•Методы, которые принадлежат интерфейсу, а не его экземплярам.

•Используются для создания вспомогательных методов (утилит).

```
interface Calculator {
    int add(int a, int b);
    default int substract(int a, int b){
        return a - b;
    }
    static int multiply(int a, int b){
        return a * b;
    }
}
 class SimpleCalculator implements Calculator{
    @Override
    public int add(int a, int b){
          return a + b;
      }
    }
    public static void main(String[] args) {
        Calculator calculator = new SimpleCalculator();
```

```
        System.out.println(calculator.add(5, 3));
        System.out.println(calculator.substract(5,3));
        System.out.println(Calculator.multiply(5,3));
    }
```

- **Java 9:**

- **`private` методы:**
- Методы с реализацией в интерфейсе, доступные только изнутри интерфейса (default и static методов).
- Позволяют разделять код дефолтных и статических методов.
- **`private static` методы:**
- Методы, которые являются приватными и статическими

```
interface  StringUtils {
    default String toUpperCase(String str){
        return prepareString(str).toUpperCase();
    }
    default String toLowerCase(String str){
        return prepareString(str).toLowerCase();
    }

    private String prepareString(String str){
        if(str == null || str.isEmpty()){
            return "";
        }
        return str.trim();
    }

    static String defaultMessage(){
        return prepareDefaultMessage();
    }
    private static String prepareDefaultMessage(){
        return "Default message: no value passed";
    }
}
class StringUtilsImpl implements StringUtils{
}
    public static void main(String[] args) {
        StringUtils utils = new StringUtilsImpl();
        System.out.println(utils.toUpperCase("  test  "));
        System.out.println(utils.toLowerCase(" TEST "));
        System.out.println(StringUtils.defaultMessage());
    }
```

- **Применение:**

- **`default`:** Добавление новых методов в интерфейсы без нарушения старого кода.
- **`static`:** Вспомогательные методы для работы с интерфейсами.
- **`private`:** Разделение кода в интерфейсах.
- **`private static`:** Вспомогательные приватные методы для статических методов.

61. Интерфейс `Comparable`. Как интерфейс `Comparable` используется для сравнения объектов? Реализация метода `compareTo()` и его роль в сортировке. Примеры работы с интерфейсом.

•Интерфейс `Comparable`:

•Определен в пакете `java.lang`.
•Используется для определения естественного порядка сравнения объектов.
•Класс реализует `Comparable`, если его объекты можно сравнивать друг с другом.
•Метод `compareTo()`:

•Абстрактный метод из интерфейса `Comparable`.
•Возвращает:
•Отрицательное число: если `this` объект меньше `other` объекта.
•Ноль: если `this` объект равен `other` объекту.
•Положительное число: если `this` объект больше `other` объекта.
•Используется для сортировки (например, `Collections.sort()`, `Arrays.sort()`).
•Примеры:

```
class Point implements Comparable<Point> {
    int x;
   int y;
   public Point(int x, int y){
       this.x = x;
       this.y = y;
   }

   @Override
   public int compareTo(Point other) {
       if(this.x != other.x){
           return Integer.compare(this.x, other.x);
       } else {
           return Integer.compare(this.y, other.y);
       }
   }
   @Override
   public String toString(){
       return "Point{x=" + x + ", y=" + y + "}";
   }
}

    public static void main(String[] args) {
        Point p1 = new Point(10, 20);
       Point p2 = new Point(5, 30);
        Point p3 = new Point(10, 10);
       System.out.println("Compare p1 with p2: " + p1.compareTo(p2)); // p1>p2
(no x)
       System.out.println("Compare p2 with p1: " + p2.compareTo(p1)); //p2 < p1
       System.out.println("Compare p1 with p3: " + p1.compareTo(p3)); //p1 > p3
(no y)

       List<Point> points = new ArrayList<>();
```

```
    points.add(p1);
        points.add(p2);
        points.add(p3);
    Collections.sort(points);
        System.out.println("Sorted list " + points); // Сортировка по x, потом по
y
    }
```

62. Интерфейс `Comparable` для классов стандартной библиотеки JAVA. Как реализован интерфейс `Comparable` в классах `String`, `Integer` и `Date`? Примеры сравнения объектов с помощью метода `compareTo()`.

- **Реализация `Comparable` в стандартных классах:**

- `String`: Сравнение строк в лексикографическом порядке (по Unicode коду).
- `Integer`, `Double`, `Long`, `Float`, `Short`, `Byte`: Сравнение числовых значений.
- `Date`: Сравнение дат (по времени).
- **Примеры сравнения:**

```
import java.util.Date;
    public static void main(String[] args) {
        // String
    String str1 = "apple";
        String str2 = "banana";
    String str3 = "apple";
    System.out.println("Compare str1 to str2: " + str1.compareTo(str2)); // <0
    System.out.println("Compare str2 to str1: " + str2.compareTo(str1)); // >0
        System.out.println("Compare str1 to str3: " + str1.compareTo(str3)); // =0


        // Integer
        Integer num1 = 10;
        Integer num2 = 5;
        Integer num3 = 10;
        System.out.println("Compare num1 to num2: " + num1.compareTo(num2)); //
>0
    System.out.println("Compare num2 to num1: " + num2.compareTo(num1)); // <0
    System.out.println("Compare num1 to num3: " + num1.compareTo(num3)); // =0

    // Date
        Date date1 = new Date(2024-1900, 1, 1);
        Date date2 = new Date(2023-1900, 12, 31);
         Date date3 = new Date(2024-1900, 1, 1);
        System.out.println("Compare date1 to date2 " + date1.compareTo(date2));
// >0
        System.out.println("Compare date2 to date1 " + date2.compareTo(date1));
// <0
        System.out.println("Compare date1 to date3 " + date1.compareTo(date3));
// =0

    }
```

63. Интерфейс `Comparable` для пользовательских классов. Как реализовать интерфейс `Comparable` для пользовательских классов? Примеры сравнения объектов на основе пользовательских критериев.

- **Реализация `Comparable`:**

- Класс должен реализовывать интерфейс `Comparable<T>`, где `T` - тип объектов, которые будут сравниваться.
- Необходимо переопределить метод `compareTo(T other)`.
- Реализация метода `compareTo()` должна соответствовать логике сравнения объектов (на основе пользовательских критериев).
- **Примеры:**

```
class Book implements Comparable<Book> {
  String title;
  String author;
  int year;

  public Book(String title, String author, int year) {
    this.title = title;
    this.author = author;
    this.year = year;
  }

@Override
  public int compareTo(Book other) {
    // сравнение по году, если одинаков - по автору, если и автор одинаковый, то по названию
    if (this.year != other.year) {
      return Integer.compare(this.year, other.year);
      } else if (!this.author.equals(other.author)){
          return this.author.compareTo(other.author);
      } else {
          return this.title.compareTo(other.title);
      }
    }

  @Override
  public String toString() {
    return "Book{title='" + title + "', author='" + author + "', year=" + year + '}';
    }
}


import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
    public static void main(String[] args) {
            Book book1 = new Book("Java", "John Doe", 2020);
            Book book2 = new Book("Python", "Jane Doe", 2022);
            Book book3 = new Book("Java for Beginners", "John Doe", 2020);
      List<Book> books = new ArrayList<>();
    books.add(book1);
    books.add(book2);
```

```
        books.add(book3);

        Collections.sort(books); // Сортировка по году, потом по автору и названию
        System.out.println(books); // Вывод списка отсортированных книг
    }
```

64. Интерфейс `Cloneable`. Понятие клонирования объектов. Как интерфейс `Cloneable` позволяет клонировать объекты? Ограничения и примеры использования.

•Клонирование объектов:

•Создание точной копии существующего объекта.
•Нужно, когда вы хотите создать независимую копию объекта, а не ссылку на один и тот же объект.

•Интерфейс `Cloneable`:

•Маркерный интерфейс (не содержит методов).
•Указывает, что объекты класса можно клонировать (поддерживается клонирование).
•Классы, поддерживающие клонирование должны реализовывать этот интерфейс, иначе будет выброшено исключение `CloneNotSupportedException`.

•Ограничения:

•**`clone()`:** Реализацию метода `clone()` нужно брать из `Object`, унаследовать и переопределить.
•**Поверхностное копирование:** Метод `clone()` по умолчанию создает поверхностную копию объекта (примитивы копируются по значению, ссылки на другие объекты — по ссылке).
•Для глубокого копирования нужно реализовывать клонирование каждого поля.

•Примеры использования:

```
class Address {
    String city;
    String street;
    public Address(String city, String street){
        this.city = city;
        this.street = street;
    }

    @Override
    public String toString() {
        return "Address{" + "city='" + city + '\'' + ", street='" + street +
'\'' + '}';
    }
}

class User implements Cloneable {
    String name;
    int age;
```

```
    Address address;

    public User(String name, int age, Address address){
        this.name = name;
        this.age = age;
        this.address = address;
    }
     @Override
    public User clone() throws CloneNotSupportedException {
        User cloned = (User) super.clone(); // Создание поверхностной копии
         cloned.address = new Address(this.address.city, this.address.street);
// глубокое клонирование
        return cloned;
     }
     @Override
    public String toString(){
            return "User{name='" + name + "', age=" + age + ", address=" +
address + "}";
    }
}


public static void main(String[] args) {
      Address address = new Address("New York", "Main street");
        User user1 = new User("John", 30, address);
        try {
        User user2 = user1.clone();
        System.out.println("User 1 " + user1);
        System.out.println("User 2 " + user2);
            user2.address.city = "Boston";
            System.out.println("User 1 " + user1); // Address User 1 не
изменился
            System.out.println("User 2 " + user2);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
```

65. Метод `clone()`. Как метод `clone()`, определенный в классе `Object`, используется совместно с интерфейсом `Cloneable`? Примеры работы с клонируемыми объектами.

- **Метод `clone()` в классе `Object`:**
- `protected` метод, создает поверхностную копию объекта.
- Вызывает `CloneNotSupportedException`, если класс не реализует `Cloneable`.
- Переопределяя метод `clone()` и вызывая через `super.clone()`, мы создаем копию объекта.
- **Использование с `Cloneable`:**
- Интерфейс `Cloneable` является маркером, указывая JVM на то что объекты данного класса можно клонировать.
- Класс должен реализовать `Cloneable` и переопределить метод `clone()` в классе, чтобы разрешить клонирование.

•Реализация метода `clone()` должна вызывать `super.clone()`, чтобы создать копию объекта, затем можно скопировать поля объекта.

•**Примеры:** (как в предыдущем примере)

```
class Address {
  String city;
  String street;
   public Address(String city, String street){
       this.city = city;
       this.street = street;
   }
    @Override
   public String toString() {
       return "Address{" + "city='" + city + '\'' + ", street='" + street + '\''
+ '}';
   }
  }

class User implements Cloneable {
   String name;
   int age;
   Address address;

  public User(String name, int age, Address address){
       this.name = name;
       this.age = age;
       this.address = address;
   }
    @Override
   public User clone() throws CloneNotSupportedException {
       User cloned = (User) super.clone(); // Создание поверхностной копии
        cloned.address = new Address(this.address.city, this.address.street); //
глубокое клонирование
       return cloned;
    }
    @Override
   public String toString(){
          return "User{name='" + name + "', age=" + age + ", address=" + address
+ "}";
    }
}


public static void main(String[] args) {
     Address address = new Address("New York", "Main street");
       User user1 = new User("John", 30, address);
       try {
       User user2 = user1.clone();
       System.out.println("User 1 " + user1);
       System.out.println("User 2 " + user2);
           user2.address.city = "Boston";
           System.out.println("User 1 " + user1); // Address User 1 не изменился
           System.out.println("User 2 " + user2);
       } catch (CloneNotSupportedException e) {
           e.printStackTrace();
       }
   }
```

66. Интерфейсы и абстрактные классы. Основные различия между интерфейсами и абстрактными классами.

- **Интерфейсы:**
- Определяют контракт (набор методов), но не содержат реализации.
- Классы могут реализовывать несколько интерфейсов (`implements`).
- Все методы по умолчанию являются `public abstract` (кроме `default` и `static`).
- Описывают что должны делать классы, но не как.
- Могут содержать только `public static final` константы.
- **Абстрактные классы:**
- Могут содержать как абстрактные, так и конкретные методы.
- Классы могут наследовать только один абстрактный класс (`extends`).
- Могут содержать поля и конструкторы.
- Описывают что и как должны делать подклассы.
- Нельзя создавать объекты абстрактных классов.
- **Основные различия:**
- **Наследование:** Класс может реализовывать несколько интерфейсов, но может наследовать только один класс (абстрактный или нет).
- **Реализация:** Интерфейсы не могут содержать реализацию методов (кроме `default` и `static`), абстрактные классы могут.
- **Поля:** Интерфейсы не могут содержать поля, абстрактные классы могут.
- **Конструкторы:** Интерфейсы не могут содержать конструкторы, абстрактные классы могут.
- **Назначение:** Интерфейсы для определения общего поведения (контракта), абстрактные классы для создания общих базовых классов.
- **Когда использовать:**
- **Интерфейсы:** Когда нужно определить контракт для разных классов.
- **Абстрактные классы:** Когда нужно предоставить общий базовый класс, но не для создания объектов абстрактного класса.

67. Понятие абстрактных классов в Java. Что такое абстрактный класс, и как он используется для создания общего базового поведения? Чем отличается абстрактный класс от интерфейса? Примеры объявления и реализации абстрактного класса с абстрактными и конкретными методами.

- **Абстрактный класс:**

- Класс, который не может быть инстанцирован (нельзя создать его объект).
- Может содержать абстрактные (`abstract`) и конкретные методы.
- Используется как базовый класс, чтобы предоставить общее поведение для подклассов.

•Может содержать поля и конструкторы.
•Если класс содержит хоть один абстрактный метод, то класс тоже должен быть объявлен абстрактным.
•Абстрактные методы:

•Методы без реализации (`abstract void methodName();`).
•Подклассы должны переопределить абстрактные методы.
•Конкретные методы:

•Методы с реализацией.
•Могут быть переопределены в подклассах.
•Различие от интерфейса:

•Интерфейс: Определяет контракт, не содержит реализации.
•Абстрактный класс: Может содержать реализацию для некоторых методов и полей.
•Классы могут реализовывать несколько интерфейсов, но наследовать только один абстрактный класс.
•Примеры:

```
abstract class Shape {
  String color;
  public Shape(String color) {
    this.color = color;
  }
   abstract double area(); // абстрактный метод
    public void draw(){ //конкретный метод
       System.out.println("Drawing a shape with color " + color);
  }

}

  class Circle extends Shape {
     double radius;
   public Circle (String color, double radius){
      super(color);
      this.radius = radius;
    }
      @Override
     public double area(){
           return Math.PI * radius * radius;
       }
  }
    class Rectangle extends Shape{
      double width;
     double height;
      public Rectangle(String color, double width, double height){
           super(color);
        this.width = width;
          this.height = height;
      }
         @Override
```

```
    public double area(){
        return width * height;
    }
}

public static void main(String[] args) {
//  Shape shape = new Shape(); // ошибка нельзя создать абстрактный класс
    Shape shape1 = new Circle("red", 5);
    shape1.draw(); // Drawing a shape with color red
    System.out.println("Area of circle: " + shape1.area()); // Area of circle: 78.53
    Shape shape2 = new Rectangle("blue", 5, 10);
    shape2.draw(); // Drawing a shape with color blue
    System.out.println("Area of rectangle: " + shape2.area()); // Area of rectangle: 50
}
```

68. Понятие абстрактных классов в Java. Объявление абстрактных методов. Что такое абстрактный метод, и какие правила нужно соблюдать при его объявлении? Как абстрактные методы помогают подклассам реализовать специфическое поведение? Примеры реализации абстрактных методов в наследуемых классах.

•Абстрактный метод:

•Метод, который объявлен в абстрактном классе и не имеет реализации (нет тела).
•Используется для объявления метода в базовом классе, но его реализация делегируется подклассам.
•Подклассы, которые наследуют абстрактный класс, обязаны переопределить (реализовать) все его абстрактные методы, иначе подкласс тоже должен быть объявлен абстрактным.

•Правила объявления:

•Используется ключевое слово `abstract`.
•Не имеет тела (заканчивается точкой с запятой `;`).
•Должен быть объявлен в абстрактном классе.
•Может иметь любой модификатор доступа (`public`, `protected`, или `package-private` - если не указываете, то по умолчанию `package-private`).

•Специфическое поведение:

•Абстрактные методы позволяют подклассам реализовывать специфическое поведение, которое отличается от базового.
•Базовый класс задает контракт (определение методов), а подклассы предоставляют реализацию.

•Примеры реализации:

```
abstract class Shape {
  String color;
  public Shape(String color) {
    this.color = color;
  }
  // Абстрактный метод
  abstract double area();
  public abstract void draw(); // Абстрактный метод

  public void printColor(){
    System.out.println("Shape color " + color);
  }
}

class Circle extends Shape {
    double radius;
    public Circle(String color, double radius){
       super(color);
        this.radius = radius;
    }
     @Override
    double area() {
    return Math.PI * radius * radius;
   }
 @Override
    public void draw(){
         System.out.println("Drawing a circle with color " + color + ", area " +
area());
   }

}

class Rectangle extends Shape {
  double width;
  double height;
  public Rectangle(String color, double width, double height) {
    super(color);
    this.width = width;
    this.height = height;
  }

   @Override
 double area() {
    return width * height;
    }
   @Override
  public void draw(){
       System.out.println("Drawing a rectangle with color " + color + ", area "
+ area());
   }

}

public static void main(String[] args) {
    Shape circle = new Circle("red", 5);
  circle.draw();
 Shape rectangle = new Rectangle("blue", 10, 5);
    rectangle.draw();
   }
```

69. Понятие абстрактных классов в Java. Особенности работы с абстрактными классами. Почему абстрактные классы нельзя инстанцировать? Как использовать абстрактный класс как основу для других классов? Примеры создания иерархии классов с базовым абстрактным классом.

•Особенности абстрактных классов:

•Нельзя создать объект абстрактного класса.
•Используется для создания общих базовых классов, которые предоставляют общую функциональность и структуру подклассам.
•Могут содержать как абстрактные, так и конкретные методы.
•Могут иметь поля и конструкторы (но не для создания объекта класса).

•Почему нельзя инстанцировать?

•Абстрактные классы не являются полными, они содержат абстрактные методы, которые должны быть реализованы в подклассах.
•Если бы можно было создавать объекты абстрактного класса, то можно было бы вызвать не реализованные (абстрактные) методы.

•Использование как основы:

•Абстрактный класс предоставляет базовую структуру и общую функциональность.
•Подклассы наследуют от абстрактного класса и реализуют специфическое поведение, переопределяя абстрактные методы.
•Создается иерархия классов, где абстрактный класс является корнем иерархии.

•Примеры иерархии:

```
abstract class Vehicle {
    String model;
    String color;
    public Vehicle(String model, String color){
        this.model = model;
        this.color = color;
    }
    abstract void start();
    abstract void stop();
    public void printInfo(){
        System.out.println("Vehicle model " + model + ", color " + color);
    }
}
class Car extends Vehicle{
    public Car(String model, String color){
        super(model, color);
    }
    @Override
    void start() {
        System.out.println("Car engine starting");
    }
    @Override
    void stop() {
        System.out.println("Car engine stopping");
```

```
    }
}

class Bike extends Vehicle {
    public Bike (String model, String color){
        super(model, color);
    }
    @Override
    void start() {
            System.out.println("Bike engine starting");
        }
    @Override
    void stop() {
            System.out.println("Bike engine stopping");
    }
}

    public static void main(String[] args) {
    //  Vehicle vehicle = new Vehicle() // Cannot be instantiated

    Vehicle car = new Car("BMW", "black");
    car.start();
        car.printInfo();
        car.stop();
    Vehicle bike = new Bike("Honda", "red");
        bike.start();
        bike.printInfo();
    bike.stop();
    }
```

70. Ограничение множественного наследования в JAVA. Множественное наследование интерфейсов. Как классы наследуют методы от нескольких интерфейсов.

•Ограничение множественного наследования:

•Java не поддерживает множественное наследование классов (нельзя создать класс, который наследует от двух или более классов).
•Это ограничение было введено для избежания проблем (например, ромбовидное наследование, коллизия методов, неоднозначность).

•Множественное наследование интерфейсов:

•Класс может реализовывать несколько интерфейсов (с помощью `implements` с перечислением через запятую).
•Поддерживается множественное наследование типа.

•Наследование методов:

•Класс наследует абстрактные методы всех интерфейсов, которые он реализует, и должен предоставить реализацию для всех абстрактных методов.
•Класс наследует дефолтные и статические методы интерфейсов.

•При конфликте методов (один и тот же дефолтный метод в разных интерфейсах) нужно переопределить метод и выбрать, какой метод использовать.

•**Пример:**

```
interface Flyable {
    void fly();

    default void printWings(){
        System.out.println("Has wings");
    }
}
interface Swimmable {
    void swim();
    default void printWings(){
        System.out.println("Has no wings");
    }
}

class Duck implements Flyable, Swimmable{
    @Override
    public void fly() {
        System.out.println("Duck flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck swimming");
    }
    @Override
    public void printWings(){
        Flyable.super.printWings(); // вызов printWings из Flyable interface
    }
}

public static void main(String[] args) {
    Duck duck = new Duck();
    duck.fly();
    duck.swim();
    duck.printWings(); // Has wings
}
```

71. Интерфейсы в Java. Особенности интерфейсов. Интерфейсы и полиморфизм. Как интерфейсы способствуют реализации полиморфизма?

•**Интерфейсы:** (повторение)

•Определяют контракт (набор методов).

•Не содержат реализации (кроме `default` и `static` методов).

•Классы могут реализовывать несколько интерфейсов.

•**Особенности:**

•Множественное наследование типа (класс может реализовывать несколько интерфейсов).

•Слабая связь (уменьшение зависимостей между классами).

•Гибкость и расширяемость (классы могут иметь различную реализацию, но соответствуют одному контракту).

•**Интерфейсы и полиморфизм:**

•Объект любого класса, реализующего интерфейс, может быть присвоен переменной типа этого интерфейса (ссылочный тип - интерфейс).

•Позволяет использовать различные объекты как объекты общего интерфейса.

•Метод будет вызываться у того объекта, на который ссылается переменная.

•**Как интерфейсы способствуют полиморфизму:**

•**Контракт:** Интерфейсы обеспечивают единый контракт, что делает возможным использование объектов разных классов, имеющих общие методы.

•**Динамическое связывание:** В момент выполнения программы определяется конкретный метод, который нужно вызвать.

•**Гибкость:** Добавление новых классов реализующих интерфейс, не нарушая существующий код.

•**Пример:** (похож на предыдущие)

```
interface Shape {
    void draw();
}
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a square");
    }
}

public static void main(String[] args) {
    Shape shape1 = new Circle(); // полиморфизм
    Shape shape2 = new Square(); // полиморфизм
    shape1.draw();  // Drawing a circle
    shape2.draw(); // Drawing a square

    // Массив
    Shape[] shapes = new Shape[2];
    shapes[0] = new Circle();
    shapes[1] = new Square();
    for(Shape shape: shapes){
        shape.draw(); // полиморфный вызов
    }

}
```

72. Обработка исключительных ситуаций в JAVA. Основные способы и подходы к обработке исключительных ситуаций в JAVA. Иерархия классов исключений в Java. Понятие и структура иерархии исключений. Чем отличаются классы `Error`, `Exception` и `RuntimeException`?

•Исключительные ситуации (Exceptions):

•События, которые нарушают нормальный ход выполнения программы (ошибки, сбои).
•Обработка исключений (exception handling) позволяет программе не завершаться аварийно, а перехватывать и обрабатывать ошибки.
•Основные способы обработки:

•**`try-catch`:** Перехват и обработка исключения в блоке `catch`.
•**`throws`:** Указание, что метод может генерировать исключение.
•**Создание собственных исключений:** Для обработки специфических ошибок в приложении.
•Иерархия классов исключений:

•`Throwable` - родительский класс для всех ошибок и исключений.
•`Error`: Представляет серьезные ошибки, которые не могут быть обработаны программой (ошибки JVM, ошибки при выделении памяти и т.д.).
•`OutOfMemoryError`, `StackOverflowError`
•`Exception`: Представляет исключения, которые могут быть перехвачены и обработаны программой (проверяемые (checked) и непроверяемые (unchecked)).
•`IOException`, `SQLException`, `ClassNotFoundException`, `NullPointerException`, `ArithmeticException`, `IndexOutOfBoundsException`
•`RuntimeException`: Непроверяемые исключения. Относятся к ошибкам программирования (ошибки времени выполнения).
*`NullPointerException`, `IndexOutOfBoundsException`, `IllegalArgumentException`, `ClassCastException`
•Отличия:

•**`Error`:** Серьезные, не обрабатываются программой.
•**`Exception`:** Обрабатываются программой (`try-catch`).
•Проверяемые (checked) исключения (`IOException`, `SQLException`) - нужно обязательно обрабатывать или пробрасывать в сигнатуре метода.
•Непроверяемые (unchecked) исключения (наследуются от `RuntimeException`) - не нужно обрабатывать явно.
•**`RuntimeException`:** Непроверяемые исключения (ошибки времени выполнения), также обрабатываются с помощью `try-catch` по желанию.

73. Создание и генерация исключений. Как создавать и генерировать исключения с помощью ключевого слова `throw`? Различия между `throw` и `throws`. Примеры создания пользовательских исключений.

•Создание и генерация исключений:

•`throw`: Используется для генерации исключения (создается объект исключения и выбрасывается).

•`throws`: Используется в сигнатуре метода для указания, что метод может генерировать определенное исключение (не используется для генерации исключения).

•`throw`:

```
public void checkAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age must be non-negative");
        }
        System.out.println("Age is valid");
}
```

•`throws`:

```
public void readFile(String filename) throws IOException {
    File file = new File(filename);
    FileReader fileReader = new FileReader(file);
    // read file...
  }
```

•Создание пользовательских исключений:

•Создать класс, наследующий от `Exception` (проверяемое) или `RuntimeException` (непроверяемое).

•Добавить конструкторы (чаще всего с сообщением).

•Примеры:

```
// Пользовательское проверяемое исключение
class InvalidEmailException extends Exception {
  public InvalidEmailException(String message) {
    super(message);
  }
}

// Пользовательское непроверяемое исключение
class NegativeBalanceException extends RuntimeException {
    public NegativeBalanceException(String message) {
        super(message);
    }
}
  public static void main(String[] args) {
        try {
            validateEmail("invalid-email");
       } catch (InvalidEmailException e){
```

```
            System.out.println("Exception catch " + e.getMessage());
        }
        try {
            withdraw(-50);
        } catch (NegativeBalanceException e){
            System.out.println("Exception catch " + e.getMessage());
        }
    }
    public static void validateEmail(String email) throws InvalidEmailException{
        if(!email.contains("@"))
        throw new InvalidEmailException("Invalid email format");
    }
    public static void withdraw(int amount){
        if(amount < 0)
        throw new NegativeBalanceException("Withdraw amount cannot be
negative");
    }
```

- **`throw` vs `throws`:**

- `throw`: для генерации исключения.
- `throws`: для указания, что метод может сгенерировать исключение, но не для генерации.

74. Обработка исключений. Структура блока `try-catch`. Как обрабатывать исключения с использованием блоков `try-catch`? Примеры обработки нескольких исключений и упорядочения блоков `catch`. Роль объекта исключения (`Exception e`) в блоке `catch`.

- **`try-catch`:**

- `try` блок: Код, в котором может возникнуть исключение.
- `catch` блок: Код для обработки исключения (выполняется, если в `try` блоке возникло исключение).
- **Обработка исключений:**

- Поместить код, где может возникнуть исключение, в `try` блок.
- Добавить один или несколько блоков `catch` для обработки конкретных исключений.
- В блоке `catch` указывается тип исключения (`Exception e`) - это объект исключения (можно получить информацию об исключении).
- **Обработка нескольких исключений:**

- Можно использовать несколько блоков `catch` для обработки разных исключений.
- Порядок блоков `catch` имеет значение: сначала должны идти блоки для более узких исключений, затем для более общих. Если исключение было поймано в одном из блоков, то дальше обработка не идет.
- **Примеры:**

```
public static void main(String[] args) {
  try{
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[5]); // вызовет IndexOutOfBoundsException

     String str = null;
   System.out.println(str.length()); //вызовет NullPointerException
  } catch (ArrayIndexOutOfBoundsException e){ //блок для
IndexOutOfBoundsException
      System.out.println("Array index out of bounds " + e.getMessage());
  } catch(NullPointerException e){ //блок для NullPointerException
      System.out.println("Null pointer exception " + e.getMessage());
  } catch (Exception e){ //общий блок для остальных исключений
       System.out.println("General exception: " + e.getMessage());
  }
    System.out.println("Program continues");
  }
```

•**Роль `Exception e`:**

•`e` - объект исключения, содержит информацию об исключении (например, сообщение об ошибке `e.getMessage()`).
•Можно использовать для логирования, отображения сообщения, и для других действий.

75. Обработка исключений. Структура блока `try-catch`. Блок `finally` и его использование. Основные причины использования. Примеры использования.

•**`finally` блок:**

•Опциональный блок, который следует за `try` и `catch` блоками.
•Выполняется всегда (вне зависимости от того, было ли исключение в `try` блоке или нет).
•Используется для освобождения ресурсов, закрытия соединений, и т.д.
•**Основные причины использования:**

•**Освобождение ресурсов:** Закрытие открытых файлов, соединений с БД, сокетов и т.д., чтобы избежать утечек ресурсов.
•**Выполнение обязательного кода:** Код, который должен быть выполнен всегда, даже если произошло исключение (например, логирование).
•**Безопасность:** Гарантия того, что ресурсы будут освобождены.
•**Примеры:**

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public static void main(String[] args) {
    FileReader fileReader = null;
       try {
```

```
        File file = new File("test.txt");
         fileReader = new FileReader(file);
        char[] chars = new char[100];
         fileReader.read(chars);
          System.out.println("Reading from the file");
    } catch(IOException e){
          System.out.println("IO exception catch " + e.getMessage());
       } finally {
          if(fileReader != null){
              try{
                  fileReader.close();
                   System.out.println("File reader closed");
              } catch (IOException e){
                  System.out.println("Error closing reader");
          }
       }
    }
   System.out.println("Program finished");
}
```

76. Обработка исключений. Пропагирование исключений. Как исключения передаются вверх по стеку вызовов? Примеры использования ключевого слова `throws` в сигнатуре методов.

•Пропагирование исключений:

•Если исключение не обрабатывается в методе, где оно произошло, то оно передается (пропагируется) вверх по стеку вызовов (в вызывающий метод).
•Если вызывающий метод не обрабатывает исключение, то оно передается еще выше, и так далее, пока не будет обработано (в конечном итоге может дойти до JVM, которая прекратит программу).
•Проверяемые исключения (`checked exceptions`) должны быть либо обработаны, либо проброшены с помощью `throws`.
•Ключевое слово `throws`:

•Указывает в сигнатуре метода, что метод может сгенерировать определенное исключение.
•Это указывает вызывающему методу, что он должен либо обработать исключение (через `try-catch`), либо также пробросить исключение (через `throws`).
•Примеры:

```
 import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public static void main(String[] args) {
       try {
          readFile("non-existent.txt");
    } catch (IOException e){
       System.out.println("IO Exception catch main " + e.getMessage());
   }
}
```

```
 public static void readFile(String filename) throws IOException {
    File file = new File(filename);
   FileReader fileReader = new FileReader(file); // может сгенерировать
исключение FileNotFoundException, который наследуется от IOException
    char[] chars = new char[100];
    fileReader.read(chars); // может сгенерировать исключение IOException
}
```

В этом примере, `readFile` пробрасывает `IOException`, и метод `main` перехватывает это исключение.

77. Обработка исключений. Проверяемые и непроверяемые исключения. Какие исключения считаются проверяемыми (checked), а какие - непроверяемыми (unchecked)? Примеры работы с ними. Исключения в популярных фреймворках. Почему большинство исключений в современных фреймворках являются непроверяемыми?

•Проверяемые (Checked) исключения:

•Исключения, которые нужно обрабатывать в коде (либо перехватывать через `try-catch`, либо пробрасывать через `throws`).
•Компилятор проверяет, что эти исключения обрабатываются.
•Наследуются от `Exception`, но не от `RuntimeException`.
•Обычно связаны с внешними ресурсами (файлы, сеть, база данных).
•Примеры: `IOException`, `SQLException`, `ClassNotFoundException`, `FileNotFoundException`.

•Непроверяемые (Unchecked) исключения:

•Исключения, которые не нужно обрабатывать явно (компилятор не проверяет).
•Наследуются от `RuntimeException` или `Error`.
•Обычно связаны с ошибками программирования (неправильные аргументы, `NullPointerException`, `ArrayIndexOutOfBoundsException`).
•Примеры: `NullPointerException`, `ArithmeticException`, `IllegalArgumentException`, `IndexOutOfBoundsException`.

•Примеры работы:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public static void main(String[] args) {
  try{
        readFile("test.txt"); // Проверяемое исключение нужно обрабатывать
        connectToDatabase(); // Проверяемое исключение нужно обрабатывать
        String str = null;
```

```
        System.out.println(str.length()); // Непроверяемое исключение не нужно
обрабатывать, но можно.
        } catch (IOException e){
            System.out.println("IO Exception " + e.getMessage());
        } catch (SQLException e) {
            System.out.println("SQL Exception " + e.getMessage());
        } catch (NullPointerException e){ //Обработка непроверяемого исключения
            System.out.println("Null pointer exception: " + e.getMessage());
    }

}
 public static void readFile(String filename) throws IOException {
        File file = new File(filename);
        FileReader reader = new FileReader(file); //checked exception
        //...
        reader.close();
    }
  public static void connectToDatabase() throws SQLException{
      Connection connection = DriverManager.getConnection("url", "user",
"password"); // checked exception
    //...
        connection.close();
    }
```

•Исключения во фреймворках:

•Большинство исключений в современных фреймворках являются непроверяемыми.

•Причины:

•Упрощение кода: Меньше `try-catch`, больше читаемости.

•Предотвращение избыточной обработки: Часто нет смысла обрабатывать `NullPointerException`, лучше исправить ошибку в коде.

•Фреймворки часто используют DI/AOP для обработки исключений глобально.

•Многие фреймворки реализуют перехват исключений и их конвертацию в исключения фреймворка.

78. Обработка исключений. Использование `try-with-resources`. Как она упрощает управление ресурсами? Примеры работы.

•`try-with-resources`:

•Упрощенный способ управления ресурсами (файлами, потоками, соединениями) при обработке исключений.

•Автоматически закрывает ресурсы после использования (в блоке `finally`).

•Работает с классами, которые реализуют интерфейс `AutoCloseable` (или `Closeable`).

•Упрощение управления ресурсами:

•Не нужно явно закрывать ресурс в блоке `finally`.

•Меньше кода, меньше ошибок (нет утечек ресурсов).

•Примеры:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public static void main(String[] args) {
    // try-with-resources
    try(FileReader reader = new FileReader("test.txt"); // Ресурс автоматически закрывается
        BufferedReader bufferedReader = new BufferedReader(reader))
    {
         String line;
        while ((line = bufferedReader.readLine()) != null){
            System.out.println(line);
        }
    } catch(IOException e){
       System.out.println("IO Exception " + e.getMessage());
    }
    System.out.println("Program finished");
  }
```

В этом примере `FileReader` и `BufferedReader` автоматически закрываются после выполнения блока `try`, даже если произошло исключение.

79. Обработка исключительных ситуаций в JAVA. Роль JVM в обработке исключений. Как JVM управляет исключениями, если они не были обработаны? Примеры поведения при неперехваченных исключениях.

•Роль JVM:

•JVM отслеживает исключения во время выполнения программы.
•Если исключение не перехвачено в `try-catch` блоке, оно пропагируется вверх по стеку вызовов.
•Если исключение доходит до главного метода (main) и там не перехвачено, то JVM его обрабатывает.

•Обработка неперехваченных исключений:

•JVM выводит сообщение об ошибке (stack trace) на консоль.
•JVM завершает программу (завершение работы текущего потока).
•При этом, могут выполнятся finally блоки.

•Примеры поведения:

```
 public static void main(String[] args) {
        System.out.println("Program starts");
    method1();
        System.out.println("Program ends"); // Не будет выведено, так как исключение не обработано
    }

    public static void method1() {
        method2();
    }

    public static void method2() {
    throw new ArithmeticException("Division by zero");
```

```
}
```

В этом примере:

- `method2()` генерирует `ArithmeticException`.
- `method1()` не перехватывает это исключение, оно пропагируется вверх.
- `main()` не перехватывает, JVM выводит стек вызовов и завершает программу.
- Сообщение “Program ends” не будет выведено.

```
public static void main(String[] args) {
    System.out.println("Program starts");
    try{
        method1();
    } catch(ArithmeticException e){
        System.out.println("Exception in main " + e.getMessage());
    }
    System.out.println("Program ends"); // Выводится при обработке исключения
}

  public static void method1() {
    method2();
  }

public static void method2() {
      throw new ArithmeticException("Division by zero");
}
```

В этом примере:

- `method2()` генерирует `ArithmeticException`.
- `method1()` не перехватывает исключение, оно пропагируется.
- `main()` перехватывает исключение, сообщение об ошибке выводится на консоль, и программа продолжается.
- Сообщение “Program ends” будет выведено.

80. Перечисления (enums) в Java. Что такое перечисления и как они используются для создания фиксированных наборов значений? Характеристики перечислений. Перечисления и типобезопасность. Примеры их применения.

- **Перечисления (Enums):**
- Специальный тип данных, который представляет собой фиксированный набор именованных констант (значений).
- Используются для создания типа данных, который может принимать только одно из заданных значений.
- Повышают читаемость и безопасность кода.
- **Характеристики:**
- Типобезопасные (нельзя присвоить неверное значение).

•Могут содержать поля, методы, конструкторы.
•Могут реализовывать интерфейсы.
•Каждая константа является объектом этого типа.
•Типобезопасность:
•Компилятор проверяет, что переменные перечислимого типа принимают только допустимые значения.
•Снижается риск ошибок, связанных с использованием неверных констант (например, вместо строк).
•Примеры применения:

```
enum Status {
  PENDING, IN_PROGRESS, COMPLETED, CANCELLED
}

enum Day {
  MONDAY("weekday"), TUESDAY("weekday"), WEDNESDAY("weekday"),
  THURSDAY("weekday"), FRIDAY("weekday"), SATURDAY("weekend"),
SUNDAY("weekend");
  String type;

Day(String type){
    this.type = type;
  }
 public String getType(){
    return type;
}
}


public static void main(String[] args) {
  Status orderStatus = Status.IN_PROGRESS;

   if (orderStatus == Status.IN_PROGRESS){
         System.out.println("Order is in progress");
   }


 for(Day day: Day.values()){
     System.out.println(day + ", type: " + day.getType());
   }
}
```

81. GUI в Java. Что такое GUI (графический пользовательский интерфейс)? Основные пакеты для работы с GUI в Java: AWT и Swing.

•GUI (Graphical User Interface):
•Графический пользовательский интерфейс.
•Способ взаимодействия пользователя с программой через графические элементы (окна, кнопки, меню).
•Более интуитивный и удобный, чем консольный интерфейс.
•Основные пакеты для GUI:

•**AWT (Abstract Window Toolkit):**
•Оригинальная библиотека для создания GUI в Java.
•Отображает элементы GUI, используя нативные компоненты операционной системы.
•Менее гибкий и менее производительный.
•Ограниченный набор компонентов.
•Компоненты тяжелые (тяжеловесные), занимают больше ресурсов.
•**Swing:**
•Более современная и гибкая библиотека для создания GUI.
•Отображает элементы GUI, используя Java-реализации (не зависят от операционной системы).
•Более гибкий, больше настраиваемый.
•Большой набор компонентов.
•Компоненты легкие (легковесные), занимают меньше ресурсов.
•Основан на AWT, но предоставляет больше возможностей.
•**Другие библиотеки:** * JavaFX - более современная UI библиотека.

82. GUI в Java. Структура GUI в JAVA при реализации через Swing и AWT. Компоненты GUI. Какие элементы составляют графический интерфейс? Примеры кнопок, текстовых полей и других компонентов.

•**Структура GUI:**

•**Контейнеры:** Компоненты, которые могут содержать другие компоненты (окна, панели).
•**Компоненты:** Элементы интерфейса (кнопки, текстовые поля, надписи, списки, и т.д.).
•**Менеджеры компоновки:** Определяют, как размещаются компоненты внутри контейнера.
•**Основные компоненты GUI:**

•**Контейнеры (Swing):**
•`JFrame`: Основное окно приложения.
•`JPanel`: Панель для группировки компонентов.
•`JDialog`: Диалоговое окно.
•**Компоненты (Swing):**
•`JButton`: Кнопка.
•`JLabel`: Надпись (текстовая метка).
•`JTextField`: Однострочное текстовое поле.
•`JTextArea`: Многострочное текстовое поле.
•`JComboBox`: Выпадающий список.

•`JList`: Список.

•`JCheckBox`: Чекбокс.

•`JRadioButton`: Радиокнопка.

•`JMenuBar`, `JMenu`, `JMenuItem`: Меню.

•**Структура AWT и Swing:** * AWT: Компоненты опираются на нативный код. * Swing: Компоненты нарисованы на Java, являются более гибкими.

•**Примеры:**

```
import javax.swing.*;
import java.awt.*;

 public class SimpleSwingApp {
         public static void main(String[] args) {
             JFrame frame = new JFrame("Simple Swing App"); //Окно
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //закрытие окна
завершает приложение
             frame.setSize(300, 200); //размер окна

            JPanel panel = new JPanel(); //панель
             panel.setLayout(new FlowLayout()); //менеджер компоновки
            // компоненты
             JButton button = new JButton("Click Me"); //кнопка
            JLabel label = new JLabel("Hello, Swing!"); //надпись
              JTextField textField = new JTextField(20); //текстовое поле

            panel.add(button); // добавление компонента на панель
             panel.add(label); // добавление надписи
           panel.add(textField); // добавление текстового поля

             frame.getContentPane().add(panel); //добавление панели на окно

            frame.setVisible(true); // отображение окна
        }
    }
```

83. AWT (Abstract Window Toolkit). Что такое AWT и как он используется для создания GUI? Примеры простых интерфейсов с использованием AWT.

•**AWT (Abstract Window Toolkit):**

•Первая библиотека Java для создания GUI.

•Использует нативные компоненты операционной системы.

•Компоненты “тяжеловесные”.

•Менее гибкий и настраиваемый, чем Swing.

•**Использование AWT для создания GUI:**

•Создание `Frame` (окна).

•Добавление компонентов (`Button`, `Label`, `TextField`).

•Управление размещением компонентов с помощью менеджеров компоновки.

•Обработка событий (действий пользователя).

•**Примеры простых интерфейсов:**

```
import java.awt.*;
public class SimpleAWTApp {
        public static void main(String[] args) {
            Frame frame = new Frame("Simple AWT App");
           frame.setSize(300, 200);
          frame.setLayout(new FlowLayout());

          Button button = new Button("Click Me"); //кнопка AWT
            Label label = new Label("Hello, AWT!"); // надпись AWT
        TextField textField = new TextField(20); // текстовое поле AWT

            frame.add(button); //добавление на фрейм
           frame.add(label);
            frame.add(textField);

            frame.setVisible(true);
    }
}
```

Этот пример создает окно с кнопкой, надписью и текстовым полем с использованием AWT. Обратите внимание, что тут все компоненты из пакета java.awt.

84. Swing в Java. Как Swing расширяет возможности AWT? Примеры создания интерфейсов с использованием Swing. Паттерн MVC в Swing. Как Swing реализует модель MVC (Model-View-Controller)? Примеры разделения логики, представления и управления в интерфейсе.

•**Swing:**

•Более современная и гибкая библиотека для GUI.
•Основана на AWT, но использует “легковесные” Java-компоненты.
•Предоставляет больше возможностей для настройки внешнего вида и поведения.

•**Расширение возможностей AWT:**

•Больший набор компонентов (`JButton`, `JLabel`, `JTextField`, `JTable`, `JTree`, и т.д.).
•Более гибкая настройка внешнего вида (look and feel).
•Поддержка MVC.
•Больше возможностей для рисования и анимации.
•Более надежна и переносима.

•**Примеры интерфейсов:**

```
import javax.swing.*;
import java.awt.*;
public class SimpleSwingApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple Swing App");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.setSize(400, 300);
```

```
        JPanel panel = new JPanel(new BorderLayout()); // менеджер компоновки BorderLayout
        JButton button1 = new JButton("North");
        JButton button2 = new JButton("South");
        JLabel label = new JLabel("Center Label", SwingConstants.CENTER); //компонент с выравниванием по центру
        panel.add(button1, BorderLayout.NORTH);
        panel.add(button2, BorderLayout.SOUTH);
        panel.add(label, BorderLayout.CENTER);


        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }
}
```

•Паттерн MVC в Swing:

•Model: Представляет данные (например, данные из базы данных, результаты вычислений).

•View: Представление данных (визуальные компоненты, графические элементы).

•Controller: Управляет взаимодействием между Model и View (обработчики событий).

•Swing неявно реализует MVC.

•Компоненты Swing (например, `JTable`, `JList`, `JComboBox`) поддерживают паттерн MVC.

•Разделение логики, представления и управления:

•Model: Отдельные классы для представления данных.

•View: Компоненты Swing для отображения данных.

•Controller: Слушатели событий (`ActionListener`, `MouseListener`) для обработки действий пользователя.

85. Структура GUI в Java. Основные компоненты GUI в Swing: контейнеры (`JFrame`, `JPanel`, `JDialog`), компоненты (`JButton`, `JLabel`, `JTextField`) и менеджеры компоновки.

•Основные компоненты GUI в Swing (повторение):

•Контейнеры:

•`JFrame`: Основное окно приложения.

•`JPanel`: Панель для группировки компонентов.

•`JDialog`: Диалоговое окно.

•Компоненты:

•`JButton`: Кнопка.

•`JLabel`: Надпись (текстовая метка).

•`JTextField`: Однострочное текстовое поле.

•`JTextArea`: Многострочное текстовое поле.

•`JComboBox`: Выпадающий список.

•`JList`: Список.

•`JCheckBox`: Чекбокс.

•`JRadioButton`: Радиокнопка.

•`JMenuBar`, `JMenu`, `JMenuItem`: Меню.

•**Менеджеры компоновки:**

•Определяют, как размещаются компоненты внутри контейнера.

•`FlowLayout`, `BorderLayout`, `GridLayout`, `BoxLayout`, `GridBagLayout` и др.

86. Класс `JFrame`. Что такое окно `JFrame`, и как использовать его для создания графического интерфейса? Примеры добавления элементов через метод `getContentPane()`.

•**`JFrame`:**

•Основное окно приложения Swing.

•Контейнер верхнего уровня (не может быть вложен в другой контейнер).

•Содержит заголовок, кнопки закрытия, минимизации и максимизации.

•**Использование:**

•Создать объект `JFrame`.

•Задать размер, заголовок, поведение при закрытии.

•Добавить компоненты на панель содержимого (`getContentPane()`).

•Отобразить окно (`setVisible(true)`).

•**`getContentPane()`:**

•Метод, который возвращает панель содержимого (`java.awt.Container`) `JFrame`.

•Компоненты добавляются на панель содержимого.

•**Примеры:**

```
import javax.swing.*;
import java.awt.*;
  public class JFrameExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My Frame");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

    JPanel panel = new JPanel(new FlowLayout()); // панель для компонентов
    JButton button = new JButton("Button");
        JLabel label = new JLabel("Label");
         panel.add(button);
         panel.add(label);

    // Добавление компонентов через getContentPane
         frame.getContentPane().add(panel);
```

```
        frame.setVisible(true);
    }
}
```

87. Класс `JPanel`. Как панель `JPanel` используется для группировки и управления компонентами? Примеры изменения менеджера компоновки с помощью метода `setLayout()`.

- **`JPanel`:**

- Контейнер для группировки компонентов.
- Используется для создания более сложных интерфейсов.
- Позволяет организовать компоненты в группы, чтобы управлять их размещением и внешним видом.
- Компонент для вкладывания в другие контейнеры.

- **Группировка компонентов:**

- Можно добавлять несколько компонентов на `JPanel`.
- Можно создавать вложенные панели.

- **`setLayout()`:**

- Метод, устанавливающий менеджер компоновки для панели.
- Позволяет изменять, как компоненты будут расположены на `JPanel` (например, `FlowLayout`, `BorderLayout`, `GridLayout`).

- **Примеры:**

```
import javax.swing.*;
import java.awt.*;
public class JPanelExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JPanel Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        // JPanel с FlowLayout
        JPanel panel1 = new JPanel();
        panel1.setLayout(new FlowLayout()); // FlowLayout
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        panel1.add(button1);
        panel1.add(button2);

        // JPanel с BorderLayout
        JPanel panel2 = new JPanel();
        panel2.setLayout(new BorderLayout()); //BorderLayout
        JLabel label = new JLabel("Center Label", SwingConstants.CENTER);
        JButton button3 = new JButton("North");
        panel2.add(label, BorderLayout.CENTER);
        panel2.add(button3, BorderLayout.NORTH);


        frame.getContentPane().add(panel1, BorderLayout.NORTH);
```

```
        frame.getContentPane().add(panel2, BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

88. Менеджеры компоновки в Java. Роль менеджеров компоновки в управлении размещением компонентов. Примеры использования менеджеров `FlowLayout`, `BorderLayout`, `GridLayout`.

•Менеджеры компоновки (Layout Managers):

•Классы, которые управляют размещением и размерами компонентов внутри контейнера.

•Обеспечивают переносимость GUI на разные платформы.

•Снимают ответственность с программиста за ручное управление позиционированием и размерами компонентов.

•Основные менеджеры:

•**`FlowLayout`:** Размещает компоненты слева направо, перенося их на следующую строку.

•**`BorderLayout`:** Размещает компоненты в пяти областях: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`.

•**`GridLayout`:** Размещает компоненты в сетку с заданным количеством строк и столбцов.

•Примеры:

```
import javax.swing.*;
import java.awt.*;
public class LayoutManagersExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layout Managers Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        // FlowLayout
        JPanel panel1 = new JPanel(new FlowLayout());
        panel1.add(new JButton("Button 1"));
        panel1.add(new JButton("Button 2"));
        panel1.add(new JButton("Button 3"));

        // BorderLayout
        JPanel panel2 = new JPanel(new BorderLayout());
        panel2.add(new JButton("North"), BorderLayout.NORTH);
        panel2.add(new JButton("South"), BorderLayout.SOUTH);
        panel2.add(new JLabel("Center", SwingConstants.CENTER),
BorderLayout.CENTER);

        // GridLayout
        JPanel panel3 = new JPanel(new GridLayout(3, 2)); // 3 rows 2 cols
        panel3.add(new JButton("Button 4"));
        panel3.add(new JButton("Button 5"));
```

```
            panel3.add(new JButton("Button 6"));
            panel3.add(new JButton("Button 7"));
            panel3.add(new JButton("Button 8"));
            panel3.add(new JButton("Button 9"));

        // добавление панелей в контейнер фрейма с помощью другого менеджера BorderLayout
        frame.getContentPane().add(panel1, BorderLayout.NORTH);
        frame.getContentPane().add(panel2, BorderLayout.CENTER);
        frame.getContentPane().add(panel3, BorderLayout.SOUTH);
        frame.setVisible(true);
    }
}
```

89. Менеджер `FlowLayout`. Как работает `FlowLayout`? Примеры настройки выравнивания и промежутков между компонентами.

- **`FlowLayout`:**

- Размещает компоненты слева направо, в порядке их добавления.
- Если компоненты не помещаются в одной строке, они переносятся на следующую строку.
- Компоненты имеют свой предпочитаемый размер.
- **Выравнивание:**

- `FlowLayout` поддерживает три типа выравнивания:
- `FlowLayout.LEFT`: Компоненты выравниваются по левому краю.
- `FlowLayout.CENTER`: Компоненты выравниваются по центру.
- `FlowLayout.RIGHT`: Компоненты выравниваются по правому краю.
- Выравнивание задается при создании объекта `FlowLayout`.
- **Промежутки (Hgap, Vgap):**

- `hgap`: Горизонтальный промежуток между компонентами.
- `vgap`: Вертикальный промежуток между строками компонентов.
- Промежутки задаются при создании объекта `FlowLayout` или через методы `setHgap()` и `setVgap()`.
- **Примеры:**

```
import javax.swing.*;
import java.awt.*;
public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FlowLayout Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        // FlowLayout с выравниванием по центру
        JPanel panel1 = new JPanel(new FlowLayout(FlowLayout.CENTER, 20, 10)); // center, hgap = 20, vgap = 10
```

```
        panel1.add(new JButton("Button 1"));
        panel1.add(new JButton("Button 2"));
        panel1.add(new JButton("Button 3"));

        // FlowLayout с выравниванием по левому краю
        JPanel panel2 = new JPanel(new FlowLayout(FlowLayout.LEFT));
        panel2.add(new JButton("Button 4"));
        panel2.add(new JButton("Button 5"));
        panel2.add(new JButton("Button 6"));

        // FlowLayout с выравниванием по правому краю
        JPanel panel3 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        panel3.add(new JButton("Button 7"));
        panel3.add(new JButton("Button 8"));
        panel3.add(new JButton("Button 9"));

        frame.getContentPane().add(panel1, BorderLayout.NORTH);
        frame.getContentPane().add(panel2, BorderLayout.CENTER);
        frame.getContentPane().add(panel3, BorderLayout.SOUTH);
        frame.setVisible(true);
    }
}
```

В этом примере, все компоненты выравнены в строке в соответствии с выбранным выравниванием и промежутками между компонентами.

90. Менеджеры компоновки в Java. Роль менеджеров компоновки в управлении размещением компонентов. Примеры использования менеджеров `FlowLayout`, `BorderLayout`, `GridLayout`.

- **Менеджеры компоновки (повторение):**

- Классы, которые управляют размещением компонентов внутри контейнера.
- Обеспечивают переносимость GUI.
- **`FlowLayout`:** (уже обсуждали) * Компоненты слева направо, переносятся на следующую строку, если не вмещаются.

- Выравнивание: `FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`
- **`BorderLayout`:** (уже обсуждали)

- Компоненты в 5 областях: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`.
- **`GridLayout`:** (уже обсуждали)

- Компоненты в сетку с заданным количеством строк и столбцов.
- **Примеры:** (повторение примера ранее)

```
import javax.swing.*;
import java.awt.*;
public class LayoutManagersExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layout Managers Example");
```

```
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(400, 300);

            // FlowLayout
            JPanel panel1 = new JPanel(new FlowLayout());
            panel1.add(new JButton("Button 1"));
            panel1.add(new JButton("Button 2"));
            panel1.add(new JButton("Button 3"));

            // BorderLayout
            JPanel panel2 = new JPanel(new BorderLayout());
            panel2.add(new JButton("North"), BorderLayout.NORTH);
            panel2.add(new JButton("South"), BorderLayout.SOUTH);
            panel2.add(new JLabel("Center", SwingConstants.CENTER),
BorderLayout.CENTER);

            // GridLayout
            JPanel panel3 = new JPanel(new GridLayout(3, 2)); // 3 rows 2 cols
            panel3.add(new JButton("Button 4"));
            panel3.add(new JButton("Button 5"));
            panel3.add(new JButton("Button 6"));
            panel3.add(new JButton("Button 7"));
            panel3.add(new JButton("Button 8"));
            panel3.add(new JButton("Button 9"));

        // добавление панелей в контейнер фрейма с помощью другого менеджера
BorderLayout
            frame.getContentPane().add(panel1, BorderLayout.NORTH);
            frame.getContentPane().add(panel2, BorderLayout.CENTER);
            frame.getContentPane().add(panel3, BorderLayout.SOUTH);
            frame.setVisible(true);
        }
    }
```

91. Менеджер `FlowLayout`. Как работает `FlowLayout`? Примеры настройки выравнивания и промежутков между компонентами.

- **`FlowLayout` (повторение):**

- Размещает компоненты слева направо в порядке добавления.
- Компоненты переносятся на следующую строку, если не помещаются в текущей.
- Компоненты имеют свой предпочитаемый размер.

- **Выравнивание (Alignment):**

- `FlowLayout.LEFT`: Компоненты выравниваются по левому краю.
- `FlowLayout.CENTER`: Компоненты выравниваются по центру.
- `FlowLayout.RIGHT`: Компоненты выравниваются по правому краю.
- Задается в конструкторе или через `setAlignment()`.

- **Промежутки (Gaps):**

•`hgap`: Горизонтальный промежуток (horizontal gap) между компонентами (в пикселях).

•`vgap`: Вертикальный промежуток (vertical gap) между строками (в пикселях).

•Задаются в конструкторе или через `setHgap()` и `setVgap()`.

•**Примеры:**

```
import javax.swing.*;
import java.awt.*;

public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FlowLayout Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        // FlowLayout с выравниванием по центру и промежутками 20x10
        JPanel panel1 = new JPanel(new FlowLayout(FlowLayout.CENTER, 20, 10));
        panel1.add(new JButton("Button 1"));
        panel1.add(new JButton("Button 2"));
        panel1.add(new JButton("Button 3"));

        // FlowLayout с выравниванием по левому краю
        JPanel panel2 = new JPanel();
        FlowLayout flowLayout = new FlowLayout(FlowLayout.LEFT);
         flowLayout.setHgap(5); // промежуток 5
          flowLayout.setVgap(5); // промежуток 5
        panel2.setLayout(flowLayout);
        panel2.add(new JButton("Button 4"));
        panel2.add(new JButton("Button 5"));
        panel2.add(new JButton("Button 6"));

        // FlowLayout с выравниванием по правому краю
         JPanel panel3 = new JPanel();
        FlowLayout flowLayout2 = new FlowLayout(FlowLayout.RIGHT);
        panel3.setLayout(flowLayout2);
         panel3.add(new JButton("Button 7"));
         panel3.add(new JButton("Button 8"));
         panel3.add(new JButton("Button 9"));



        frame.getContentPane().add(panel1, BorderLayout.NORTH);
        frame.getContentPane().add(panel2, BorderLayout.CENTER);
        frame.getContentPane().add(panel3, BorderLayout.SOUTH);


        frame.setVisible(true);
    }
}
```

92. Менеджер `BorderLayout`. Как `BorderLayout` делит контейнер на регионы (`NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`)? Примеры создания интерфейсов с четкой организацией областей.

•`BorderLayout`:

•Делит контейнер на пять областей: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`.

•Каждая область может содержать только один компонент.

•Области `NORTH` и `SOUTH` растягиваются по горизонтали, `EAST` и `WEST` по вертикали.

•`CENTER` область занимает всё оставшееся пространство.

•**Организация областей:**

•`BorderLayout.NORTH`: Верхняя область.

•`BorderLayout.SOUTH`: Нижняя область.

•`BorderLayout.EAST`: Правая область.

•`BorderLayout.WEST`: Левая область.

•`BorderLayout.CENTER`: Центральная область.

•**Примеры:**

```
import javax.swing.*;
import java.awt.*;
public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BorderLayout Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        JPanel panel = new JPanel(new BorderLayout());

        JButton northButton = new JButton("NORTH");
        JButton southButton = new JButton("SOUTH");
        JButton eastButton = new JButton("EAST");
        JButton westButton = new JButton("WEST");
        JLabel centerLabel = new JLabel("CENTER", SwingConstants.CENTER);
        // Добавление компонентов в конкретные области
        panel.add(northButton, BorderLayout.NORTH);
        panel.add(southButton, BorderLayout.SOUTH);
        panel.add(eastButton, BorderLayout.EAST);
        panel.add(westButton, BorderLayout.WEST);
        panel.add(centerLabel, BorderLayout.CENTER);

        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }
}
```

93. Менеджер `GridLayout`. Как компоненты размещаются в сетке с использованием `GridLayout`? Примеры создания таблиц или форм.

•**`GridLayout`:**

•Размещает компоненты в сетку (таблицу) с заданным количеством строк и столбцов.

•Компоненты располагаются в ячейках сетки, занимая всё доступное пространство.

•Все ячейки имеют одинаковый размер.

•Компоненты добавляются в порядке слева направо, сверху вниз.

•**Создание таблиц/форм:**

•Идеален для создания простых таблиц, форм, клавиатур.

•**Примеры:**

```
import javax.swing.*;
import java.awt.*;

public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
       JPanel panel = new JPanel(new GridLayout(3, 3)); // 3 строки, 3 столбца

        // добавление кнопок
       for (int i=1; i<= 9; i++){
           panel.add(new JButton("Button " + i));
       }

        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }
}
```

94. Менеджер `BoxLayout`. Как компоненты размещаются по горизонтали или вертикали с помощью `BoxLayout`? Примеры последовательного расположения элементов.

•**`BoxLayout`:**

•Размещает компоненты в одну строку (горизонтально) или один столбец (вертикально).

•Компоненты располагаются последовательно в одном измерении.

•Позволяет более гибко управлять размером компонентов, чем `FlowLayout`.

•Не выравнивает компоненты по вертикали.

•**Горизонтальное/вертикальное:**

•`BoxLayout.X_AXIS`: Горизонтальное расположение.

•`BoxLayout.Y_AXIS`: Вертикальное расположение.

•Задается в конструкторе при создании объекта `BoxLayout`.

•**Примеры:**

```
import javax.swing.*;
import java.awt.*;

public class BoxLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BoxLayout Example");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(400, 300);
```

```
        // Горизонтальное расположение
        JPanel panel1 = new JPanel();
        panel1.setLayout(new BoxLayout(panel1, BoxLayout.X_AXIS));
        panel1.add(new JButton("Button 1"));
        panel1.add(new JButton("Button 2"));
        panel1.add(new JButton("Button 3"));

        // Вертикальное расположение
        JPanel panel2 = new JPanel();
        panel2.setLayout(new BoxLayout(panel2, BoxLayout.Y_AXIS));
        panel2.add(new JButton("Button 4"));
        panel2.add(new JButton("Button 5"));
        panel2.add(new JButton("Button 6"));



        frame.getContentPane().add(panel1, BorderLayout.NORTH);
        frame.getContentPane().add(panel2, BorderLayout.CENTER);
        frame.setVisible(true);
    }
}
```

95. Границы в Swing. Как использовать границы для улучшения внешнего вида интерфейса? Примеры применения границ.

- **Границы (Borders):**
- Классы для добавления рамок, отступов и других визуальных эффектов вокруг компонентов.
- Улучшают визуальную привлекательность и читаемость интерфейса.
- Определены в пакете `javax.swing.border`.
- **Использование границ:**
- Создать объект границы (например, `LineBorder`, `EtchedBorder`, `TitledBorder`).
- Установить границу для компонента через метод `setBorder()`.
- **Примеры:**

```
import javax.swing.*;
 import javax.swing.border.Border;
import javax.swing.border.EtchedBorder;
 import javax.swing.border.LineBorder;
 import javax.swing.border.TitledBorder;
  import java.awt.*;
  public class BordersExample {
      public static void main(String[] args) {
          JFrame frame = new JFrame("Borders Example");
          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          frame.setSize(400, 300);

          JPanel panel1 = new JPanel();
          panel1.setLayout(new FlowLayout());

          Border lineBorder = new LineBorder(Color.BLACK, 2); //  стиль border
          JButton button1 = new JButton("Line Border");
          button1.setBorder(lineBorder); //установка border
          panel1.add(button1);
```

```
        JPanel panel2 = new JPanel();
        panel2.setLayout(new FlowLayout());
        Border etchedBorder = new EtchedBorder();
        JButton button2 = new JButton("Etched Border");
        button2.setBorder(etchedBorder);
        panel2.add(button2);

        JPanel panel3 = new JPanel();
        panel3.setLayout(new FlowLayout());
        Border titledBorder = new TitledBorder("Titled Border");
        JButton button3 = new JButton("Titled Border");
        button3.setBorder(titledBorder);
        panel3.add(button3);

        frame.getContentPane().add(panel1, BorderLayout.NORTH);
        frame.getContentPane().add(panel2, BorderLayout.CENTER);
        frame.getContentPane().add(panel3, BorderLayout.SOUTH);

        frame.setVisible(true);
    }
}
```

- **Виды границ:**
- `LineBorder`: Граница в виде линии.
- `EtchedBorder`: Граница в виде утопленной или выпуклой линии.
- `TitledBorder`: Граница с заголовком.
- `EmptyBorder`: Граница в виде отступа.
- `CompoundBorder`: Граница из нескольких границ.

96. GUI и событийная модель в Java. Что такое событийная модель, и как она используется для взаимодействия компонентов через события? Основные элементы событийной модели.

- **Событийная модель (Event-Driven Model):**

- Модель программирования, где поток управления определяется событиями.
- Вместо последовательного выполнения кода, программа реагирует на события, генерируемые пользователем или системой.
- Основа для создания интерактивных GUI.
- **Взаимодействие компонентов:**

- Компоненты GUI (кнопки, текстовые поля, списки) генерируют события (нажатия, ввод текста, выбор элемента).
- Компоненты могут регистрировать слушателей событий (event listeners) для перехвата и обработки этих событий.
- Слушатель событий определяет, как реагировать на событие.
- **Основные элементы:**

•**Источник события (Event Source):** Компонент, который генерирует событие (например, кнопка).
•**Событие (Event):** Действие, которое произошло (нажатие кнопки, ввод текста, перемещение мыши).
•**Слушатель события (Event Listener):** Объект, который регистрируется на источнике, чтобы перехватывать события.
•**Обработчик события (Event Handler):** Метод слушателя события, который вызывается при возникновении события.

97. Обработка событий в Java. Как источник события, слушатель и обработчик взаимодействуют в событийной модели? Примеры добавления слушателей событий. Модель делегирования событий. Как работает модель делегирования событий?

•**Взаимодействие:**

1.**Источник события:** Создает объект события (например, `ActionEvent`, `MouseEvent`).
2.**Слушатель события:** Регистрируется на источнике события (через `addXXXListener()` метод) и реализует интерфейс слушателя (например, `ActionListener`, `MouseListener`).
3.**Обработчик события:** Когда происходит событие, источник уведомляет слушателей, вызывая соответствующий метод слушателя (например, `actionPerformed()`, `mouseClicked()`).
•**Добавление слушателей:**

1.Используется метод `addXXXListener()`, где `XXX` - тип слушателя (например, `addActionListener()`, `addMouseListener()`).
2.Метод принимает в качестве параметра объект, реализующий интерфейс слушателя.
•**Примеры:**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class EventHandlingExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Event Handling Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        JPanel panel = new JPanel(new FlowLayout());
      JButton button = new JButton("Click Me");

        // Добавление слушателя события
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
```

```
                System.out.println("Button clicked!"); //Обработчик
                JOptionPane.showMessageDialog(frame, "Button Clicked"); // показ
сообщения
            }
        });

        panel.add(button);
        frame.getContentPane().add(panel);
         frame.setVisible(true);
    }
}
```

•Модель делегирования событий:

1.Источник события не знает, кто будет обрабатывать событие.
2.Он делегирует обработку события зарегистрированным слушателям.
3.Это уменьшает связи между компонентами.

98. Обработка событий при реализации GUI в JAVA. Классы событий пакета `java.awt.event`. Какие классы событий предоставляет пакет `java.awt.event`? Примеры обработки событий мыши и клавиатуры.

•Пакет `java.awt.event`:

•Содержит классы и интерфейсы для работы с событиями.
•`ActionEvent`: Событие от компонентов (кнопки, меню), связанных с действием.
•`MouseEvent`: События от мыши (нажатие, перемещение, отпускание).
•`KeyEvent`: События от клавиатуры (нажатие, отпускание, ввод символа).
•`ItemEvent`: События от компонентов (выпадающие списки, чекбоксы, переключатели), связанных с изменением состояния.
•`FocusEvent`: Событие изменения фокуса компонента.
•`WindowEvent`: События от окна (`JFrame`, `JDialog`) (закрытие, активация, деактивация).
•`TextEvent`: События от текстовых полей, связанные с изменением текста.
•и др.

•Обработка событий мыши и клавиатуры:

```
import javax.swing.*;
 import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
public class EventClassesExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Event Classes Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

         JPanel panel = new JPanel(new FlowLayout());

        // Обработка событий мыши
```

```
        JLabel mouseLabel = new JLabel("Mouse events will be shown here");
        panel.add(mouseLabel);

        panel.addMouseListener(new MouseListener() {
            @Override
            public void mouseClicked(MouseEvent e) {
                mouseLabel.setText("Mouse clicked");
            }
            @Override
            public void mousePressed(MouseEvent e) {
                mouseLabel.setText("Mouse pressed");
            }
            @Override
            public void mouseReleased(MouseEvent e) {
                mouseLabel.setText("Mouse released");
            }
            @Override
            public void mouseEntered(MouseEvent e) {
                mouseLabel.setText("Mouse entered");
            }
            @Override
            public void mouseExited(MouseEvent e) {
                mouseLabel.setText("Mouse exited");
            }
        });

        // Обработка событий клавиатуры
        JLabel keyLabel = new JLabel("Key events will be shown here");
        panel.add(keyLabel);
        frame.addKeyListener(new KeyListener() {
            @Override
            public void keyTyped(KeyEvent e) {
                keyLabel.setText("Key typed " + e.getKeyChar());
            }
            @Override
            public void keyPressed(KeyEvent e) {
                keyLabel.setText("Key pressed " + e.getKeyChar());
            }
            @Override
            public void keyReleased(KeyEvent e) {
                keyLabel.setText("Key released " + e.getKeyChar());
            }
        });
        frame.setFocusable(true);
        frame.requestFocusInWindow(); //установка фокуса
        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }
}
```

В этом примере, `panel` обрабатывает события мыши и `frame` обрабатывает события клавиатуры.

99. Обработка событий мыши в JAVA. Как использовать интерфейсы `MouseListener` и `MouseMotionListener` для обработки событий мыши? Примеры обработки нажатий и перемещений.

•**Интерфейс `MouseListener`:**

•Обрабатывает события мыши, связанные с нажатием и отпусканием кнопок.

•Методы:

•`mouseClicked(MouseEvent e)`: Клик мышью (нажатие и отпускание кнопки).

•`mousePressed(MouseEvent e)`: Нажатие кнопки мыши.

•`mouseReleased(MouseEvent e)`: Отпускание кнопки мыши.

•`mouseEntered(MouseEvent e)`: Курсор мыши вошел в компонент.

•`mouseExited(MouseEvent e)`: Курсор мыши покинул компонент.

•**Интерфейс `MouseMotionListener`:**

•Обрабатывает события мыши, связанные с перемещением.

•Методы:

•`mouseMoved(MouseEvent e)`: Курсор мыши переместился над компонентом.

•`mouseDragged(MouseEvent e)`: Курсор мыши переместился с нажатой кнопкой.

•**Примеры:**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;

public class MouseEventsExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Mouse Events Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        JPanel panel = new JPanel(new FlowLayout());
        JLabel mouseLabel = new JLabel("Mouse events will be shown here");
         panel.add(mouseLabel);


        panel.addMouseListener(new MouseListener() {
            @Override
             public void mouseClicked(MouseEvent e) {
                mouseLabel.setText("Mouse clicked at " + e.getX() + ", " +
e.getY());
            }
            @Override
             public void mousePressed(MouseEvent e) {
                mouseLabel.setText("Mouse pressed at " + e.getX() + ", " +
e.getY());
            }
            @Override
             public void mouseReleased(MouseEvent e) {
                 mouseLabel.setText("Mouse released at " + e.getX() + ", " +
e.getY());
             }
            @Override
             public void mouseEntered(MouseEvent e) {
```

```
                mouseLabel.setText("Mouse entered the area");
            }
            @Override
            public void mouseExited(MouseEvent e) {
                mouseLabel.setText("Mouse exited the area");
            }
        });

        panel.addMouseMotionListener(new MouseMotionListener() {
            @Override
            public void mouseMoved(MouseEvent e) {
                mouseLabel.setText("Mouse moved to " + e.getX() + ", " +
e.getY());
            }
            @Override
            public void mouseDragged(MouseEvent e) {
                mouseLabel.setText("Mouse dragged to " + e.getX() + ", " +
e.getY());
            }
        });



        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }
}
```

100. Обработка событий клавиатуры в JAVA. Как обрабатывать события клавиатуры с использованием `KeyListener`? Примеры регистрации слушателей клавиатурных событий.

- **Интерфейс `KeyListener`:**

- Обрабатывает события, связанные с нажатием и отпусканием кнопок на клавиатуре.
- Методы:
- `keyTyped(KeyEvent e)`: Символ введен (после нажатия и отпускания).
- `keyPressed(KeyEvent e)`: Клавиша нажата.
- `keyReleased(KeyEvent e)`: Клавиша отпущена.
- **Регистрация слушателя:**

- Вызывается метод `addKeyListener()` компонента, которому нужно обрабатывать события клавиатуры.
- Нужно установить фокус на компонент для приема событий.
- **Примеры:**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
public class KeyEventsExample {
    public static void main(String[] args) {
```

```
        JFrame frame = new JFrame("Key Events Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(400, 300);

        JPanel panel = new JPanel(new FlowLayout());
        JLabel keyLabel = new JLabel("Key events will be shown here");
        panel.add(keyLabel);

        // Добавление слушателя на фрейм (JFrame)
         frame.addKeyListener(new KeyListener() {
            @Override
            public void keyTyped(KeyEvent e) {
              keyLabel.setText("Key typed: " + e.getKeyChar());
            }
          @Override
          public void keyPressed(KeyEvent e) {
             keyLabel.setText("Key pressed: " + e.getKeyChar());
          }
            @Override
          public void keyReleased(KeyEvent e) {
             keyLabel.setText("Key released: " + e.getKeyChar());
          }
      });
          frame.setFocusable(true);
           frame.requestFocusInWindow(); //установка фокуса
       frame.getContentPane().add(panel);
        frame.setVisible(true);

    }
}
```

101. Обобщённое программирование в Java. Понятие обобщённого программирования и его роль в упрощении создания алгоритмов для работы с различными типами данных. История разваития в JAVA. Примеры проектирования универсальных структур данных и алгоритмов.

•Обобщённое программирование (Generic Programming):

•Методология программирования, которая позволяет писать код, работающий с различными типами данных, без потери типобезопасности.
•Код пишется независимо от конкретных типов, работает с абстрактными типами (параметрами типа).
•Позволяет создавать универсальные алгоритмы, структуры данных, компоненты.
•Роль:

•Уменьшение дублирования кода (повторного написания кода для разных типов).
•Повышение типобезопасности (ошибки отлавливаются на этапе компиляции).
•Более читаемый и поддерживаемый код.
•Переиспользование кода.
•История развития:

•До Java 5 не было обобщений. Для работы с различными типами часто использовался `Object`, что приводило к небезопасным приведениям типов.
•Java 5 ввела Generics (обобщения), сделав код более безопасным и простым.
•Примеры:

•До обобщений: “`java import java.util.ArrayList; import java.util.List; public static void main(String[] args) { List numbers = new ArrayList(); // без generics numbers.add(1); numbers.add(2); numbers.add(“test”); // нет ошибки при компиляции for (Object num : numbers) { // int n = (Integer) num; // ClassCastException System.out.println(num); //1, 2, test }

```
}
```
```

•С обобщениями:

```
  import java.util.ArrayList;
import java.util.List;
  public static void main(String[] args) {
            List<Integer> numbers = new ArrayList<>(); // с generics
            numbers.add(1);
            numbers.add(2);
           //numbers.add("test"); // ошибка на этапе компиляции
            for (int num : numbers) {
                System.out.println(num); //1, 2
            }
    }
```

•Примеры универсальных структур данных:

•`List<T>`: Список с элементами типа `T`.

•`Set<T>`: Множество с элементами типа `T`.

•`Map<K, V>`: Словарь с ключами типа `K` и значениями типа `V`.

•Стек, очередь, дерево и др. структуры данных.

•Примеры универсальных алгоритмов:

•Сортировка, поиск, сравнение, и т.д., работающие с любыми типами, которые соответствуют необходимым условиям (например, реализуют Comparable).

102. Generics в Java. Реализация обобщенного программирования через Generics. Основные синтаксические конструкции: параметры типов, обобщенные классы и методы. Примеры работы с параметризованными классами и методами. Примущества и недостатки Generics.

•Generics (Обобщения):

•Реализация обобщенного программирования в Java.
•Позволяет создавать классы, интерфейсы и методы, которые могут работать с различными типами данных без потери типобезопасности.
```

•**Синтаксические конструкции:**

•**Параметры типов:**

•Используются угловые скобки `<T>` для определения типа-параметра.

•`T` - это имя типа, обычно заглавная буква (например, `T`, `E`, `K`, `V`).

•Можно несколько параметров `<T, K>`.

•**Обобщенные классы:**

•Классы, которые используют параметры типа при объявлении (например, `List<T>`).

•**Обобщенные методы:**

•Методы, которые используют параметры типа (например, `Collections.sort(List<T> list)`).

•**Примеры параметризованных классов и методов:**

```
    class Box<T>{
    private T content;
        public void setContent(T content){
            this.content = content;
        }
       public T getContent(){
              return content;
       }
}
  class ArrayUtils{
      public static<T> void printArray(T[] array){
           for(T item: array){
             System.out.print(item + " ");
           }
           System.out.println();
      }
  }
  public static void main(String[] args) {
        Box<Integer> intBox = new Box<>(); // Integer type
         intBox.setContent(123);
        System.out.println(intBox.getContent());
    Box<String> stringBox = new Box<>(); // String type
         stringBox.setContent("test");
         System.out.println(stringBox.getContent());

     Integer[] numbers = {1,2,3,4};
    String[] strings = {"A", "B", "C"};
    ArrayUtils.printArray(numbers); // Parametrized method with Integer Array
    ArrayUtils.printArray(strings); // Parametrized method with String Array
}
```

•**Преимущества:**

•Типобезопасность (ошибки отлавливаются на этапе компиляции).

•Уменьшение дублирования кода (переиспользование).

•Более читаемый и поддерживаемый код.

•Устраняет необходимость ручного приведения типов.

•**Недостатки:**

•Более сложный синтаксис.
•Стирание типов (type erasure) на этапе выполнения: информация о типах удаляется в процессе компиляции.

103. Коллекции и Generics в Java. Как использование Generics повысило типобезопасность коллекций, таких как `ArrayList`, `HashMap` и `HashSet`? Примеры создания и обработки коллекций с обобщениями.

•**Коллекции (Collections):**

•Классы для хранения и обработки групп объектов (например, `ArrayList`, `HashMap`, `HashSet`).
•До Java 5 коллекции хранили объекты типа `Object`, что требовало небезопасного приведения типов.
•**Generics и типобезопасность:**

•Generics ввели типы-параметры в коллекции, позволяя указать, объекты какого типа будут храниться в коллекции (например, `ArrayList<String>`, `HashMap<Integer, User>`).
•Это позволяет компилятору проверять типобезопасность на этапе компиляции, предотвращая ошибки времени выполнения, например `ClassCastException`.
•Уменьшилась необходимость явного приведения типов.
•**Примеры:**

```
 import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
public static void main(String[] args) {
    // ArrayList с обобщениями
    List<String> names = new ArrayList<>();
    names.add("Alice");
    names.add("Bob");
    //names.add(123); // Compile error
   for(String name: names){
        System.out.println(name);
    }

     // HashMap с обобщениями
    Map<String, Integer> ages = new HashMap<>();
   ages.put("Alice", 30);
      ages.put("Bob", 25);
   // ages.put(1, "test"); //Compile error
     for (Map.Entry<String, Integer> entry: ages.entrySet()){
           System.out.println("Name " + entry.getKey() + " age " +
entry.getValue());
    }
```

```
        // HashSet с обобщениями
        Set<Integer> numbers = new HashSet<>();
        numbers.add(1);
        numbers.add(2);
        //numbers.add("test"); // Compile error
        for (int num : numbers) {
            System.out.println(num);
        }
    }
```

•Преимущества:

•Компилятор проверяет, что в коллекцию добавляются объекты правильного типа.
•Безопасность - избежание ошибок приведения типа.
•Меньше кода.
•Лучшая читаемость.

104. Параметризованные методы. Понятие параметризованных методов в Java. Как они позволяют работать с любыми типами данных? Примеры реализации методов с обобщенными параметрами и их вызова.

•Параметризованные методы (Generic Methods): (повторение)

•Методы, использующие параметры типа (типы-переменные).
•Могут работать с разными типами данных, не теряя типобезопасности.
•Параметр типа объявляется перед возвращаемым типом метода `<T>`, `<K, V>`.
•Работа с любыми типами данных:

•Использование параметров типа (типы-переменные), которые во время вызова метода, заменяются конкретными типами.
•Компилятор проверяет корректность типов при вызове.
•Можно работать с примитивами (через обертки), классами, интерфейсами.
•Примеры:

```
class ArrayUtils {
    // Параметризованный метод для печати массива любого типа
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    // Параметризованный метод для поиска элемента в массиве
    public static <T> boolean contains(T[] array, T target) {
        for (T element : array) {
            if (element.equals(target)) {
                return true;
            }
```

```
        }
        return false;
    }

    // Параметризованный метод с ограничением Comparable interface
    public static <T extends Comparable<T>> T max(T[] array) {
        if (array == null || array.length == 0) {
            return null;
        }
        T max = array[0];
        for (int i = 1; i < array.length; i++) {
            if (array[i].compareTo(max) > 0) {
                max = array[i];
            }
        }
        return max;
    }
}
public static void main(String[] args) {
    Integer[] numbers = {1, 5, 2, 8, 3};
    String[] strings = {"apple", "banana", "cherry"};
    ArrayUtils.printArray(numbers);
    ArrayUtils.printArray(strings);

    System.out.println("Contains 5 in number list? " +
ArrayUtils.contains(numbers, 5));
    System.out.println("Contains apple in string list? " +
ArrayUtils.contains(strings, "apple"));

    Integer maxInt = ArrayUtils.max(numbers);
    System.out.println("Max from integer list " + maxInt);
    String maxStr = ArrayUtils.max(strings);
    System.out.println("Max from string list " + maxStr);

}
```

105. Generics в Java. Типовые ограничения в Generics. Как задать ограничения на параметры типов с помощью ключевых слов `extends` и `super`? Примеры их использования для обеспечения гибкости и безопасности обобщений.

•Типовые ограничения:

•Позволяют ограничить типы, которые могут использоваться как параметры типа.
•Используются ключевые слова `extends` и `super`.
•**`extends`:**

•Указывает, что тип должен быть подклассом указанного класса (или реализовывать интерфейс).
•`T extends Number`: Тип `T` должен быть Number или его подклассом (`Integer`, `Double`, и т.д.).
•`T extends Comparable<T>`: Тип `T` должен реализовывать интерфейс `Comparable` и может сравнивать себя с другими объектами того же типа.
•`T extends ClassA & InterfaceB`: тип должен быть подтипом и ClassA и InterfaceB.

•**`super`:**

•Указывает, что тип должен быть суперклассом указанного класса.

•`T super Integer`: Тип `T` должен быть Integer или его суперклассом (например, `Number`, `Object`).

•Используется реже.

•**Примеры:**

```
class NumberUtils {
    // ограничение extends Number
    public static <T extends Number> double sum(T[] array) {
        double sum = 0;
        for (T num : array) {
            sum += num.doubleValue();
        }
        return sum;
    }
    // ограничение extends Comparable
    public static <T extends Comparable<T>> T max(T[] array) {
        if (array == null || array.length == 0) {
            return null;
        }
        T max = array[0];
        for (int i = 1; i < array.length; i++) {
            if (array[i].compareTo(max) > 0) {
                max = array[i];
            }
        }
        return max;
    }

public static <T super Integer> void printList(T[] list){
        for(Object obj: list){
        System.out.println(obj);
    }
 }
}
public static void main(String[] args) {
        Integer[] numbers = {1, 2, 3, 4, 5};
        Double[] doubles = {1.0, 2.0, 3.0};
        String[] strings = {"apple", "banana"};

    System.out.println("Sum of numbers: " + NumberUtils.sum(numbers));
        System.out.println("Sum of doubles: " + NumberUtils.sum(doubles));
     //  NumberUtils.sum(strings); // ошибка компиляции
    Integer maxInteger = NumberUtils.max(numbers); // extends Comparable
    System.out.println("Max Integer " + maxInteger);
        //NumberUtils.max(strings); // ошибка компиляции
    Number[] numberArray = {1, 2.0, 3.5};
        NumberUtils.printList(numberArray); // super Integer
    Object[] objectArray = {1, 2.0, "Test"};
        NumberUtils.printList(objectArray);
 }
```

106. Обобщенные интерфейсы. Использование Generics для создания универсальных интерфейсов. Примеры реализации обобщенных интерфейсов и их применения в реальных задачах.

•Обобщенные интерфейсы (Generic Interfaces):

•Интерфейсы, которые используют параметры типа.
•Позволяют создавать универсальные интерфейсы, которые могут работать с разными типами.
•Классы реализуют обобщенный интерфейс, указывая конкретный тип.
•Реализация:

•Интерфейс объявляется с параметром типа `<T>`.
•Класс реализует интерфейс, указывая конкретный тип для `T` (например, `String`, `Integer`, `User`).
•Примеры:

```
// Обобщенный интерфейс
 interface Repository<T> {
    void add(T item);
     T getById(int id);
    List<T> getAll();
    }

class UserRepository implements Repository<User>{
 List<User> users = new ArrayList<>();

   @Override
     public void add(User user) {
         users.add(user);
     }
    @Override
   public User getById(int id) {
        return users.get(id);
      }
     @Override
   public List<User> getAll() {
        return users;
    }

}
   class ProductRepository implements Repository<Product>{
     List<Product> products = new ArrayList<>();
     @Override
     public void add(Product product) {
         products.add(product);
     }
     @Override
     public Product getById(int id) {
       return products.get(id);
     }
     @Override
     public List<Product> getAll() {
       return products;
```

```
    }
}

class User { String name; public User(String name){this.name = name;}}
class Product { String name; public Product(String name){this.name = name;}}


public static void main(String[] args) {
    UserRepository userRepository = new UserRepository();
    userRepository.add(new User("John"));
    userRepository.add(new User("Bob"));
    System.out.println("User name: " + userRepository.getById(0).name);

    ProductRepository productRepository = new ProductRepository();
    productRepository.add(new Product("Laptop"));
    productRepository.add(new Product("Phone"));
    System.out.println("Product name: " + productRepository.getById(0).name);
}
```

107. Generics в Java. Подстановочные знаки (Wildcards). Как использовать `?`, `<? extends T>` и `<? super T>` для работы с коллекциями? Примеры их применения.

•Подстановочные знаки (Wildcards):

•Используются в Generics, когда точный тип параметра не важен.

•Представляются символом `?`.

•`?` (Unbounded Wildcard):

•Представляет любой тип.

•`List<?>`: Список элементов любого типа.

•Полезен, если не нужно использовать конкретный тип в коде.

•Можно получить элементы, но нельзя добавить.

•`<? extends T>` (Upper-Bounded Wildcard):

•Указывает, что тип должен быть подтипом T (T или его подкласс).

•`List<? extends Number>`: Список элементов типа Number или его подклассов (`Integer`, `Double`, и т.д.).

•Чтение элементов безопасно (можно получить элемент типа Number), но добавление элементов небезопасно (компилятор не может гарантировать тип добавляемого элемента).

•`<? super T>` (Lower-Bounded Wildcard):

•Указывает, что тип должен быть суперклассом T (T или его суперкласс).

•`List<? super Integer>`: Список элементов типа Integer или его суперклассов (`Number`, `Object`).

•Добавление элементов безопасно (можно добавить элемент типа Integer), но чтение элементов небезопасно (мы можем получить элемент типа Object).

•Примеры:

```
 import java.util.ArrayList;
import java.util.List;
class ListUtils {
      public static void printList(List<?> list) { //Unbounded wildcard
          for (Object element : list) {
                System.out.print(element + " ");
           }
          System.out.println();
   }

   public static double sumList(List<? extends Number> list){ // Upper-Bounded
wildcard
      double sum = 0;
          for (Number number : list){
          sum += number.doubleValue();
    }
     return sum;
}
 public static void addInteger(List<? super Integer> list, Integer num){
//Lower-Bounded wildcard
       list.add(num);
     }
}
public static void main(String[] args) {
     List<Integer> numbers = new ArrayList<>();
     numbers.add(1);
      numbers.add(2);
     List<String> names = new ArrayList<>();
     names.add("Alice");
     names.add("Bob");

     ListUtils.printList(numbers);
      ListUtils.printList(names);

     List<Double> doubles = new ArrayList<>();
     doubles.add(1.0);
     doubles.add(2.0);
     System.out.println("Sum of doubles: " + ListUtils.sumList(doubles));
   System.out.println("Sum of numbers: " + ListUtils.sumList(numbers));

 List<Object> objects = new ArrayList<>();
       ListUtils.addInteger(objects, 5); //Lower-Bounded
 }
```

108. Generics в Java. Стирание типов (Type Erasure). Как информация о Generics удаляется во время компиляции? Примеры преобразования Generics в сырой тип.

•Стирание типов (Type Erasure):

•Процесс удаления информации о параметрах типа (Generics) во время компиляции.
•Байт-код JVM не содержит информацию о типах-параметрах.
•Компилятор проверяет типы на этапе компиляции, но во время выполнения все коллекции и классы с обобщениями имеют “сырой” тип (raw type).

•Обеспечивает обратную совместимость с кодом, написанным до Java 5 (до Generics).

•Преобразование в сырой тип:

•Типы-параметры заменяются на `Object` или на их верхние границы.

•Например, `List<String>` в байт-коде будет представлен как `List`.

•Все коллекции приходят к базовому типу - Object.

•Примеры:

```
  public static void main(String[] args) {
        List<String> names = new ArrayList<>();
       names.add("Alice");
        names.add("Bob");

        // Raw type
      List rawList = names; // raw type
     rawList.add(123); // No compile error (but can cause runtime exception)

        for(Object item : rawList) {
           System.out.println(item); // Alice, Bob, 123
        }

        List<Integer> numbers = new ArrayList<>();
        Class<?> listClass = numbers.getClass();
       System.out.println(listClass); //class java.util.ArrayList ( нем
информации о Integer )
    }
```

•Последствия стирания типов:

•Нельзя использовать оператор `instanceof` с обобщенными типами (`list instanceof List<Integer>` - ошибка).

•Нельзя создать массив из обобщенного типа (`new T[10]` - ошибка).

•Нельзя перегружать методы, которые отличаются только параметрами типа (после стирания типов).

109. Коллекции в Java. Понятие коллекций как структур данных для хранения объектов. Основные интерфейсы и классы в Java Collections Framework (JCF). Примеры использования коллекций для хранения и обработки данных.

•Коллекции (Collections):

•Структуры данных, предназначенные для хранения и управления группами объектов.

•Обеспечивают эффективное хранение, поиск, добавление и удаление объектов.

•Находятся в пакете `java.util`.

•Java Collections Framework (JCF):

•Набор интерфейсов и классов для работы с коллекциями.

•Предоставляет общую архитектуру и набор алгоритмов для работы с коллекциями.

•Основные интерфейсы:

•`Collection`: Общий интерфейс для всех коллекций.

•`List`: Упорядоченная коллекция с возможностью дублирования элементов (`ArrayList`, `LinkedList`).

•`Set`: Неупорядоченная коллекция без дубликатов (`HashSet`, `TreeSet`).

•`Map`: Коллекция для хранения пар "ключ-значение" (`HashMap`, `TreeMap`).

•Примеры использования:

```
import java.util.*;
 public static void main(String[] args) {
       // List (ArrayList)
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
         names.add("Charlie");
        System.out.println("List names " + names);

        // Set (HashSet)
        Set<Integer> numbers = new HashSet<>();
       numbers.add(1);
         numbers.add(2);
         numbers.add(1); // Дубликат не добавляется
      System.out.println("Set numbers " + numbers);

        // Map (HashMap)
       Map<String, Integer> ages = new HashMap<>();
      ages.put("Alice", 30);
       ages.put("Bob", 25);
     System.out.println("Map ages " + ages);

        // Обработка данных
      for (String name: names){
         System.out.println(name);
       }
}
```

110. Иерархия коллекций. Структура иерархии коллекций в Java. Основные интерфейсы (`Collection`, `List`, `Set`, `Map`) и их ключевые особенности. Примеры реализации различных типов коллекций.

•Иерархия коллекций:

•`java.lang.Iterable` (не интерфейс коллекций, но супер интерфейс для Collection):

•Интерфейс для итерации по коллекции.

•Метод `iterator()`: возвращает итератор.

•`java.util.Collection`:

•Базовый интерфейс для всех коллекций.

•Основные
методы: `add()`, `remove()`, `size()`, `isEmpty()`, `contains()`, `clear()`, `iterator()` и др.
•Интерфейсы-наследники: `List`, `Set`, `Queue`
•`java.util.List`:
•Упорядоченная коллекция.
•Разрешает дубликаты.
•Индексированный доступ.
•Реализации: `ArrayList`, `LinkedList`.
•`java.util.Set`: * Неупорядоченная коллекция (зависит от реализации). * Не разрешает дубликаты.
•Реализации: `HashSet`, `TreeSet`.
•`java.util.Queue`:
•Очередь.
•Реализации: `LinkedList`, `PriorityQueue`, `ArrayDeque`.
•`java.util.Map`:
•Коллекция для хранения пар ключ-значение.
•Ключи должны быть уникальными.
•Реализации: `HashMap`, `TreeMap`, `LinkedHashMap`.
•**Основные интерфейсы и их особенности (повторение):**

•`Collection`: Общий интерфейс.
•`List`: Упорядоченная, с дубликатами, по индексу.
•`Set`: Неупорядоченная, без дубликатов.
•`Map`: Ключ-значение, ключи уникальные.
•**Примеры реализаций:**

```
 import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;

public static void main(String[] args) {
    // List
    List<String> list = new ArrayList<>();
   List<Integer> linkedList = new LinkedList<>();

      // Set
   Set<String> hashSet = new HashSet<>();
  Set<Integer> treeSet = new TreeSet<>();
```

```
    // Map
    Map<String, Integer> hashMap = new HashMap<>();
Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
    Map<String, Integer> treeMap = new TreeMap<>();
}
```

111. `LinkedList` в Java. Особенности класса `LinkedList` как реализации интерфейса `List`. Преимущества использования.

•`LinkedList`:

•Реализация интерфейса `List`.
•Основана на двусвязном списке (каждый элемент имеет ссылки на следующий и предыдущий элементы).
•Добавление и удаление в середине списка быстрее, чем в `ArrayList` (не нужно сдвигать элементы).
•Хранит элементы в порядке добавления.
•Поддерживает все операции `List`.

•Особенности:

•Двусвязный список: Каждый элемент содержит ссылки на предыдущий и следующий элементы.
•Непрерывное выделение памяти (элементы могут храниться в разных местах в памяти, не обязательно подряд).
•Оптимизирован для вставок и удалений в середине списка.
•Менее эффективный доступ к элементам по индексу (требуется обход списка).

•Преимущества использования:

•Быстрые вставки и удаления элементов в середине списка.
•Подходит для реализации стека, очереди, двунаправленной очереди (deque).
•Гибкость при работе с последовательностями элементов.

•Когда использовать:

•Когда часто нужно добавлять и удалять элементы в середине списка.
•Когда порядок элементов важен.
•Когда не требуется быстрый доступ к элементам по индексу.

112. Коллекции в Java. Понятие коллекций как структур данных для хранения объектов. Основные цели использования коллекций. Роль `Iterable` в Java Collections Framework.

•Коллекции (повторение):
•Структуры данных для хранения и управления группами объектов.
•Предоставляют эффективные способы хранения, поиска, добавления и удаления объектов.

•**Основные цели использования:**

•**Хранение данных:** Организация и хранение большого количества объектов.

•**Обработка данных:** Доступ, добавление, удаление, поиск, сортировка объектов.

•**Оптимизация:** Выбор подходящей структуры данных для конкретной задачи.

•**Переиспользование:** Готовые реализации структур данных.

•**Разделение ответственности:** Код для работы с данными и с их хранением.

•**Гибкость:** Легко добавлять новые типы данных и операции.

•**Интерфейс `Iterable`:**

•Интерфейс из пакета `java.lang`.

•Позволяет итерироваться по коллекции (перебирать элементы один за другим).

•Единственный метод:

•`Iterator<T> iterator()`: Возвращает итератор (объект, позволяющий перемещаться по коллекции).

•Все коллекции реализуют `Iterable`, что позволяет использовать расширенный цикл `for-each`.

•**Роль `Iterable`:**

•Интерфейс `Iterable` — супер-интерфейс для всех коллекций, предоставляет стандартный способ для обхода элементов.

•Поддержка расширенного цикла `for-each`.

•Возможность итерироваться через стандартные интерфейсы.

•Разделение ответственности (collection хранит данные, iterator отвечает за обход).

113. Коллекции в Java. Реализации `List` - `ArrayList`. Особенности функционирования `ArrayList`. Пример использования `ArrayList`.

•**`ArrayList`:**

•Реализация интерфейса `List`.

•Основана на массиве, размер которого может динамически изменяться (расширяется при добавлении).

•Быстрый доступ к элементам по индексу (по номеру).

•Медленные вставки и удаления в середине списка (требуется сдвиг элементов).

•Упорядоченная коллекция.

•Разрешает дубликаты.

•Хранит элементы в порядке добавления.

•**Особенности функционирования:**

•Массив: Элементы хранятся в массиве (внутри).

•Динамическое изменение размера: При добавлении элемента, когда массив заполнен, создается новый массив большего размера, и элементы копируются из старого в новый.

•Быстрый доступ по индексу: O(1) — время доступа не зависит от размера списка.
•Медленные вставки и удаления в середине списка: O(n) — время пропорционально размеру списка (требуется сдвиг элементов).

•Пример использования:

```
    import java.util.ArrayList;
import java.util.List;

  public static void main(String[] args) {
        List<String> names = new ArrayList<>();

        // Добавление элементов
       names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println("List names " + names);
        // Доступ к элементу по индексу
       String name = names.get(1);
       System.out.println("Element on index 1: " + name);

       // Изменение элемента
       names.set(1, "John");
       System.out.println("After set operation " + names);

        // Вставка элемента в середину списка
      names.add(1,"Peter");
       System.out.println("After insert operation " + names);

        // Удаление элемента по индексу
       names.remove(0);
     System.out.println("After remove operation " + names);

         // Размер списка
        int size = names.size();
        System.out.println("Size of list " + size);

         //Проверка наличия элемента
        boolean contains = names.contains("John");
       System.out.println("List contains John? " + contains);

        // Итерация по списку
      for (String n : names) {
           System.out.println(n);
        }
    }
```

•Когда использовать:

•Когда нужно часто получать элементы по индексу.
•Когда не нужно часто добавлять и удалять элементы в середине списка.
•В большинстве случаев `ArrayList` используется чаще, чем `LinkedList`.

114. Коллекции в Java. Создание Generic Collection в Java. Преимущества данного подхода. Примеры.

•**Generic Collection:**

•Коллекция (например, `List`, `Set`, `Map`), созданная с использованием обобщений (Generics) — с параметром типа `<T>`.

•Тип-параметр указывает, объекты какого типа будут храниться в коллекции (например, `List<String>`, `Set<Integer>`, `Map<String, User>`).

•**Преимущества:**

•**Типобезопасность:** Компилятор проверяет, что в коллекцию добавляются объекты правильного типа, снижая вероятность ошибок времени выполнения (например, `ClassCastException`).

•**Меньше кода:** Не нужно явного приведения типов.

•**Читаемость:** Улучшает читаемость кода, так как сразу видно, объекты какого типа хранит коллекция.

•**Переиспользование:** Одна коллекция может хранить объекты разных типов (через параметр типа), а не создавать разные коллекции для разных типов.

•**Примеры:**

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
public static void main(String[] args) {
    // Generic List
    List<String> names = new ArrayList<>();
    names.add("Alice");
    names.add("Bob");
    //names.add(1);  Compile error - mun не String
    for(String name: names){
        System.out.println(name);
    }
    // Generic Set
    Set<Integer> numbers = new HashSet<>();
    numbers.add(1);
    numbers.add(2);
    //numbers.add("test"); // Compile error - mun не Integer
    for(Integer num: numbers){
        System.out.println(num);
    }

    // Generic Map
    Map<String, User> users = new HashMap<>();
    users.put("Alice", new User("Alice"));
    users.put("Bob", new User("Bob"));
    //users.put(1, "test");  // Compile error - mun ключа не String, а значения не User
    for(Map.Entry<String, User> entry: users.entrySet()){
        System.out.println("Key " + entry.getKey() + " value " + entry.getValue().name);
    }
```

```
}

 static class User {
    String name;
      public User(String name){
          this.name = name;
      }
   }
```

115. Коллекции и Generics. Использование Generics для типобезопасности в коллекциях. Примеры создания типизированных списков и множеств.

•**Generics и типобезопасность:** (повторение)

•Generics позволяют параметризовать коллекции типом, чтобы хранить объекты только определенного типа.
•Компилятор проверяет, что в коллекцию добавляются объекты правильного типа, предотвращая `ClassCastException`.
•Уменьшает необходимость явного приведения типов при извлечении.
•**Типизированные списки и множества:**

•`List<T>`: Список, хранящий объекты типа `T`.
•`Set<T>`: Множество, хранящее объекты типа `T`.
•`ArrayList<T>`, `LinkedList<T>`, `HashSet<T>`, `TreeSet<T>`: Конкретные реализации `List` и `Set` с параметром типа `T`.
•**Примеры:**

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
  public static void main(String[] args) {
      // Типизированный список строк
       List<String> names = new ArrayList<>();
       names.add("Alice");
       names.add("Bob");
     //  names.add(1); // Ошибка компиляции
      for (String name : names) {
        System.out.println(name);
       }
      // Типизированное множество чисел
    Set<Integer> numbers = new HashSet<>();
        numbers.add(1);
       numbers.add(2);
     // numbers.add("test"); // Ошибка компиляции
       for (Integer number : numbers) {
           System.out.println(number);
       }

        // Типизированный список объектов
         List<User> users = new ArrayList<>();
         users.add(new User("John"));
        users.add(new User("Alice"));
```

```
        // users.add(1);  // Ошибка компиляции
        for(User user: users){
            System.out.println(user.name);
        }
    }
static class User {
        String name;
        public User(String name){
            this.name = name;
        }
    }
```

116. `ArrayList` в Java. Понятие `ArrayList` как реализации интерфейса `List`. Основные методы (`add`, `get`, `remove`) для работы со списками. Примеры добавления, удаления и доступа к элементам.

- **`ArrayList` (повторение):**

- Динамический массив, реализация интерфейса `List`.
- Обеспечивает быстрый доступ к элементам по индексу.
- Замедление операций вставки и удаления в середине списка.
- **Основные методы:**

- `add(E element)`: Добавляет элемент в конец списка.
- `add(int index, E element)`: Добавляет элемент в указанную позицию.
- `get(int index)`: Получает элемент по указанному индексу.
- `remove(int index)`: Удаляет элемент по указанному индексу.
- `remove(Object object)`: Удаляет первое вхождение указанного объекта.
- `set(int index, E element)`: Заменяет элемент по указанному индексу.
- `size()`: Возвращает количество элементов.
- `contains(Object object)`: Проверяет, есть ли элемент в списке.
- и другие.
- **Примеры:**

```
import java.util.ArrayList;
import java.util.List;
 public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        // Добавление элементов
        names.add("Alice");
        names.add("Bob");
          names.add("Charlie");
       System.out.println("List names " + names);

        // Доступ к элементу по индексу
       String name = names.get(1);
        System.out.println("Get element by index 1: " + name); //Bob

        // Вставка элемента
        names.add(1, "Peter");
```

```
    System.out.println("List names after insert " + names);

        // Изменение элемента
        names.set(1, "John");
        System.out.println("List names after set " + names);

        // Удаление элемента по индексу
        names.remove(0);
        System.out.println("List names after remove by index " + names);

        // Удаление элемента по значению
        names.remove("John");
        System.out.println("List names after remove by value " + names);

        // Проверка наличия элемента
        boolean contains = names.contains("Bob");
        System.out.println("List contains Bob? " + contains);
    }
```